RUN-TIME TECHNIQUES FOR DYNAMIC MULTITHREADED COMPUTATIONS

BY

VIJAY KARAMCHETI

B.Tech., Indian Institute of Technology, Kanpur, 1988
M.S., The University of Texas at Austin, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

RUN-TIME TECHNIQUES FOR DYNAMIC MULTITHREADED COMPUTATIONS

Vijay Karamcheti, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1998
Andrew A. Chien, Advisor

Programming models based on dynamic multithreading enable convenient expression of irregular parallel applications by providing high-level features such as object-oriented programming, a global object name space, and elective concurrency. Despite these programmability advantages, such models have not gained widespread use because they are challenging to implement efficiently on scalable parallel machines due to their dynamic, irregular thread structure (e.g., granularity variations) and unpredictable data access patterns.

In this dissertation, we present an execution framework consisting of novel run-time mechanisms that overcome these challenges. The framework separates concerns of local and parallel efficiency, enabling mechanisms to be independently developed and optimized for each. Local efficiency techniques, *hybrid stack-heap execution* and *pull messaging*, support low-overhead thread management and communication that deliver performance conditioned upon run-time locality and load balance. Parallel efficiency techniques, *view caching* and a *hierarchical load balancing framework*, exploit knowledge of application behavior to create good locality and load balance situations at run time, achieving performance robust over computation irregularity.

The execution framework has been implemented in the Illinois Concert System, targeting the ICC++ language. Evaluation of mechanisms on two target platforms, the Cray T3D and the SGI Origin 2000, for four large irregular applications demonstrates their individual effectiveness and collective sufficiency: Each application achieves performance comparable to the best achievable using low-level means. Additionally, these improvements persist across a range of hardware communication architecture interfaces. Our results imply that high-level expressions of parallel programs need not sacrifice on performance.

To my family,

ACKNOWLEDGMENTS

This thesis is the long-awaited culmination of seven years of graduate school at Illinois. I would like to thank everyone who made it possible.

First and foremost, I would like to thank my advisor Andrew Chien for setting a high standard, and for his continued guidance, support, and encouragement to help me reach it. Andrew has taught me a lot about research and being a researcher, and as I start my own academic career, I hope that I can match the passion he brings to everything he does. He will be a tough act to follow.

I owe a great deal to Professor Michael Loui for being a mentor to me, and to Professors Daniel Reed and Prithviraj Banerjee for their advice, on matters both technical and otherwise. I would also like to thank Professors Laxmikant Kale and Josep Torrellas for participating on my final examination committee, and for their assistance in improving various aspects of this thesis.

My officemates, both past and present, deserve special mention for contributing to my growth as a researcher. I would like to thank John Plevyak, Xingbin Zhang, Jae Kim, Julian Dolby, Scott Pakin, and the rest of the Concurrent Systems Architecture Group for countless animated discussions, valuable insights, and shared good times.

All my friends and relatives have been a constant source of support during the lengthy Ph.D. process. That I was able to complete this thesis is largely because of their encouragement. I would especially like to mention Kumar, Shyamala, Amitabh, and Sharad who refused to let me get discouraged during some very trying times.

And most of all, I would like to express my profound gratitude to my family: my parents, Sriram and Veeralakshmi, my brothers, Ravi and Arun, my in-laws, Siva and Krishnaveny, my wife, Shanthi, and my son, Siddharth for their love, support, understanding, patience, encouragement, and for constantly reminding me of what is important. You have been the source of my strength and deserve all the credit for this thesis.

TABLE OF CONTENTS

CHAPTER	PAGE
1 INTRODUCTION	1
1.1 Thesis	3
1.2 Overview	4
1.3 Contributions	8
1.4 Organization	9
2 BACKGROUND	10
2.1 Concurrent Object-Oriented Programming	10
2.1.1 Concurrency and synchronization	11
2.1.2 High-level programming features	11
2.2 Dynamic Multithreaded Execution	12
2.2.1 Execution entities	13
2.2.2 Characteristics of target platforms	14
2.2.3 Mapping entities to hardware resources	15
2.3 Illinois Concert C++ (ICC++)	16
2.3.1 Basic programming interface	16
2.3.2 Performance-optimization interfaces	17
2.3.3 Example: An ICC++ program	19
2.4 Hardware Platforms	24
2.4.1 Cray T3D	25
2.4.2 SGI Cray Origin 2000	26
2.5 The Illinois Concert System	28
2.5.1 The Concert compiler	29
2.5.2 The Concert run-time system	30
2.5.3 Base run-time system mechanisms	31
2.6 Summary	37
3 PROBLEM AND APPROACH	39
3.1 Performance Challenges	39
3.1.1 Elements of performance	39
3.1.2 Dynamic computation models	40
3.1.3 Characteristics of irregular applications	41
3.2 Problem Statement	42
3.3 Previous Work	43
3.3.1 Thread management and communication	43
3.3.2 Data locality	44

3.3.3 Load balance . . . 44
3.3.4 Summary . . . 45
3.4 Solution Approach . . . 45
3.4.1 Local efficiency . . . 46
3.4.2 Parallel efficiency . . . 46
3.5 Summary . . . 49

4 LOCAL EFFICIENCY . . . 50
4.1 Overview . . . 50
4.1.1 An ideal local efficiency building block . . . 51
4.1.2 Shortcomings of existing mechanisms . . . 51
4.2 Robust Thread Management: Hybrid Stack-Heap Execution . . . 55
4.2.1 Design and implementation . . . 55
4.2.2 Primitive costs . . . 60
4.2.3 Related work . . . 61
4.2.4 Summary . . . 62
4.3 Robust Communication: Pull Messaging . . . 63
4.3.1 Design and implementation . . . 63
4.3.2 Primitive costs . . . 65
4.3.3 Related work . . . 68
4.3.4 Summary . . . 68
4.4 Performance of Mechanisms on the Cray T3D . . . 69
4.5 Performance of Mechanisms on the SGI Origin 2000 . . . 72
4.5.1 Applicability . . . 72
4.5.2 Performance . . . 74
4.6 Summary . . . 76

5 DATA LOCALITY . . . 77
5.1 Overview . . . 77
5.1.1 The problem . . . 78
5.1.2 Overview of solution . . . 79
5.2 Experimental Context . . . 80
5.2.1 Object-consistent DSM: The C Region Library . . . 80
5.2.2 Sharing microkernel . . . 83
5.3 Object-consistent DSM: Low-Overhead Coherence . . . 87
5.3.1 DSM implementations . . . 87
5.3.2 Primitive operation costs . . . 90
5.4 Performance of Object-consistent DSM . . . 94
5.4.1 Overview of experiments . . . 94
5.4.2 Sensitivity to responsiveness . . . 95
5.4.3 Sensitivity to occupancy . . . 98
5.4.4 Summary . . . 100
5.5 View Caching: Latency-Tolerant Coherence . . . 100
5.5.1 Rationale: Synchronous DSM protocols . . . 101
5.5.2 Solution overview . . . 102
5.5.3 View caching framework . . . 103

5.5.4 Prototype view caching implementation . . . 106
5.5.5 Primitive operation costs . . . 112
5.6 Performance of View Caching . . . 114
5.6.1 Sensitivity to responsiveness . . . 115
5.6.2 Sensitivity to occupancy . . . 118
5.6.3 Summary . . . 121
5.7 Related Work . . . 121
5.8 Summary . . . 124

6 LOAD BALANCE . . . 126
6.1 Background . . . 126
6.1.1 The problem . . . 127
6.1.2 Solution overview . . . 127
6.2 Intractability of the Load-Balancing Problem . . . 128
6.2.1 Task structure in dynamic multithreaded computations . . . 128
6.2.2 Intractability results . . . 130
6.3 Load-Balancing Framework . . . 137
6.3.1 Strategy for load balancing . . . 138
6.3.2 Components of the framework . . . 139
6.4 Language and Run-Time Support . . . 140
6.4.1 Language support: Associating load balancing policies . . . 140
6.4.2 Run-time support: Synthesizing application load balance . . . 141
6.5 Library of Load-Balancing Policies . . . 143
6.5.1 Object placement . . . 144
6.5.2 Thread placement . . . 145
6.5.3 Thread ordering . . . 147
6.5.4 Summary . . . 147
6.6 Relating Application Archetypes to Policies . . . 148
6.6.1 Attributes of irregular applications . . . 148
6.6.2 A task generation microkernel . . . 149
6.6.3 Overview of experiments . . . 152
6.6.4 Microkernel performance results . . . 153
6.6.5 Suitability of load-balancing policies . . . 163
6.7 Related Work . . . 165
6.8 Summary . . . 167

7 PERFORMANCE EVALUATION . . . 169
7.1 Overview . . . 169
7.1.1 Experimental setup . . . 170
7.1.2 Structure of the study . . . 170
7.1.3 Application suite . . . 171
7.2 IC-Cedar . . . 175
7.2.1 Application structure . . . 176
7.2.2 Data locality and load-balancing optimizations . . . 178
7.2.3 Performance results . . . 179
7.3 Radiosity . . . 183

7.3.1 Application structure . . . 183
7.3.2 Data locality and load-balancing optimizations . . . 186
7.3.3 Performance results . . . 189
7.4 Gröbner . . . 197
7.4.1 Application structure . . . 197
7.4.2 Data locality and load-balancing optimizations . . . 198
7.4.3 Performance results . . . 200
7.5 Phylogeny . . . 204
7.5.1 Application structure . . . 207
7.5.2 Data locality and load-balancing optimizations . . . 208
7.5.3 Performance results . . . 211
7.6 Overall Impact of Mechanisms . . . 217
7.6.1 Application structure . . . 217
7.6.2 Data locality and load-balancing optimizations . . . 218
7.6.3 Performance impact . . . 222
7.7 Summary . . . 226

8 CONCLUSION . . . 228
8.1 Thesis Summary . . . 228
8.2 Perspective . . . 231
8.3 Future Work . . . 233

APPENDIX A RAW MICROKERNEL DATA . . . 238
A.1 Multiparty Communication Microkernel . . . 238
A.2 Communicate-Compute Microkernel . . . 239
A.3 Sharing Microkernel . . . 241
A.4 Task Generation Microkernel . . . 250

APPENDIX B RAW APPLICATION DATA . . . 257
B.1 IC-Cedar . . . 257
B.2 Radiosity . . . 263
B.3 Gröbner . . . 277
B.4 Phylogeny . . . 287

REFERENCES . . . 298

VITA . . . 312

LIST OF TABLES

Table **Page**

2.1 Performance summary of hardware platforms. 24
2.2 Costs of basic Concert thread management primitives on the T3D and the Origin. 33
2.3 Latency and software overheads of Concert communication primitives on the T3D and the Origin for various message sizes. 36

4.1 Various thread interaction schemas in the hybrid stack-heap execution model. 56
4.2 Call and fallback costs (at the caller) for different caller-callee scenarios on the Cray T3D, in terms of overhead required in addition to a C function call. 61
4.3 Base source and destination overheads of the push and pull messaging schemes on the T3D (all times in μs). 66
4.4 Call and fallback costs (at the caller) for different caller-callee scenarios on the SGI Origin 2000, in terms of overhead required in addition to a C function call. 73

5.1 Summary of the object-consistent CRL interface. 82
5.2 Summary of base CRL protocol states. 84
5.3 Summary of base CRL protocol events. 85
5.4 Values of sharing microkernel parameters. 87
5.5 Primitive object sharing operations on the Cray T3D (all costs in μs). 91
5.6 Primitive object sharing operations on the SGI Cray Origin 2000 (all costs in μs). 92
5.7 Options and values for customizing different components of coherence processing. 105
5.8 View caching extensions to the region-based DSM interface. 106
5.9 Library of view coherence types, their description, and the corresponding coherence protocol components. 108
5.10 Summary of view caching protocol states. 111
5.11 Summary of view caching protocol events. 111
5.12 Primitive object sharing operations on the Cray T3D corresponding to two view coherence types (all costs in μs). 112
5.13 Primitive object sharing operations on the SGI Cray Origin 2000 corresponding to two view coherence types (all costs in μs). 113

6.1 Library of load balancing policies. 144
6.2 Values of task generation microkernel parameters. 152

7.1 The irregular applications suite. 172
7.2 Sequential execution profile of ICC++ applications on the Cray T3D. 172
7.3 Optimizing IC-Cedar performance using view caching and load balancing frameworks. . . . 178
7.4 Optimizing radiosity performance using view caching and load balancing frameworks. . . . 187

7.5 Optimizing Gröbner performance using view caching and load-balancing frameworks. . . . 199
7.6 Optimizing phylogeny performance using view caching and load balancing frameworks. . . 209
7.7 Optimizing data locality using the view-caching framework. 219
7.8 Optimizing load balance using the load-balancing framework. 221

A.1 Source overheads (in μs) of push- and pull-messaging on the Cray T3D for various multi-party communication traffic patterns. 238
A.2 Execution efficiency on the SGI Origin 2000 for various communicate-compute microkernel configurations. 239
A.3 Execution efficiency on the Cray T3D for various communicate-compute microkernel configurations. 240
A.4 Execution times on the Cray T3D (in seconds) for various sharing microkernel configurations. 241
A.5 Execution times on the SGI Origin 2000 (in seconds) for various sharing microkernel configurations. 245
A.6 Execution times on the Cray T3D (in seconds) for various task microkernel configurations. . 250
A.7 Execution times on the SGI Origin 2000 (in seconds) for various task microkernel configurations. 253

B.1 IC-Cedar execution times (in seconds) on the Cray T3D. 257
B.2 IC-Cedar execution times (in seconds) on the SGI Origin 2000. 260
B.3 Radiosity execution times (in seconds) on the Cray T3D. 264
B.4 Radiosity execution times (in seconds) on the SGI Origin 2000. 271
B.5 Gröbner execution times (in seconds) on the Cray T3D. 277
B.6 Gröbner execution times (in seconds) on the SGI Origin 2000. 282
B.7 Phylogeny execution times (in seconds) on the Cray T3D. 287
B.8 Phylogeny execution times (in seconds) on the SGI Origin 2000. 293

LIST OF FIGURES

Figure		Page
2.1	Execution model showing processing nodes (large boxes), contexts (small boxes), objects (circles), and thread creation (dark arrows).	13
2.2	Patches (circles) and interactions (boxes) in the hierarchical radiosity application.	19
2.3	C++ expression of the hierarchical radiosity application.	21
2.4	ICC++ expression of the hierarchical radiosity application.	22
2.5	The T3D processing element.	25
2.6	The Origin 2000 processing element.	27
3.1	Spatial and temporal irregularities in implementations of dynamic multithreaded computations.	42
3.2	Four communication architecture interfaces.	47
4.1	Code structure of the communicate-compute microkernel (shown using ICC++ syntax).	52
4.2	Microkernel efficiency on the T3D and the Origin for the Base run-time system mechanisms.	54
4.3	Caller (left) and Callee (right) code sequences for the Non-blocking schema.	57
4.4	Caller (left) and callee (right) code sequences for the may-block schema.	58
4.5	May-block interaction schema.	58
4.6	Caller (left) and callee (right) code sequences for the continuation-passing schema.	59
4.7	Continuation-passing interaction schema.	60
4.8	Distributed message queue implementation of pull-messaging.	65
4.9	Source overheads for push (left) and pull (right) versions of messaging on the T3D for four message sizes.	67
4.10	Impact of local efficiency mechanisms on the Cray T3D.	69
4.11	Regions of the compute granularity and fraction of local accesses parameter space where the microkernel achieves $\geq 70\%$ efficiency for different run-time system versions on the Cray T3D.	71
4.12	Impact of run-time system mechanisms on the SGI Origin 2000.	74
4.13	Regions of the compute granularity and fraction of local accesses parameter space where the microkernel achieves $\geq 70\%$ efficiency for different run-time system versions on the Origin.	75
5.1	Traditional software DSM protocols incur increased idle time in the context of dynamic multithreaded computations.	79
5.2	Code structure of the sharing microkernel.	86

5.3 Four implementations of the CRL interface corresponding to various communication architectures shown in the context of a read miss. 88
5.4 Sensitivity to responsiveness: Microkernel execution time breakdown on the T3D and the Origin as the use granularity is varied. 96
5.5 Sensitivity to responsiveness: Microkernel execution time breakdown on the T3D and the Origin as the interrequest compute granularity is varied. 96
5.6 Sensitivity to occupancy: Microkernel execution time breakdown on the T3D and the Origin as the read-to-write ratio is varied. 98
5.7 Sensitivity to occupancy: Microkernel execution time breakdown on the T3D and the Origin as the distribution of shared accesses is varied. 99
5.8 Components of cache processing for a write miss. 104
5.9 Sensitivity to responsiveness: Microkernel execution time breakdown on the T3D and the Origin showing impact of view caching protocols as use granularity is varied. 116
5.10 Sensitivity to responsiveness: Microkernel execution time breakdown on the T3D and the Origin showing impact of view caching protocols as interrequest compute granularity is varied. 117
5.11 Sensitivity to occupancy: Microkernel execution time breakdown on the T3D and the Origin showing impact of view caching protocols as read-to-write ratio is varied. 119
5.12 Sensitivity to occupancy: Microkernel execution time breakdown on the T3D and the Origin showing impact of view caching protocols as distribution of shared accesses is varied. . . . 120
5.13 Microkernel execution time breakdown on the T3D and the Origin showing the impact of view caching protocols versus different communication architectures for a fixed set of parameters. 122

6.1 Task structure in dynamic multithreaded computations. 129
6.2 Intuition for transforming TSP into HAMMING-TSP. 133
6.3 Bit representation of HAMMING-TSP vectors. 134
6.4 Overall load balancing strategy: Different subsets of threads (shown by the white, cross-hatched, and black circles) are subject to different load-balancing policies. 138
6.5 The custom scheduler framework augments the default FIFO scheduler with a set of application specific custom schedulers, each of which manages a set of threads according to a different load-balancing policy. 142
6.6 Code structure of the task generation microkernel. 150
6.7 Microkernel execution time breakdown as the thread granularity is varied with no thread imbalance ($\vee = 0.0$). 154
6.8 Microkernel execution time breakdown as the thread granularity is varied with medium thread imbalance ($\vee = 0.5\kappa$). 155
6.9 Microkernel execution time breakdown as the thread granularity is varied with high thread imbalance ($\vee = \kappa$). 156
6.10 Microkernel execution time breakdown as the thread imbalance is varied. 157
6.11 Microkernel execution time breakdown as the locality penalty is varied. 158
6.12 Microkernel execution time breakdown as the priority penalty is varied. 159
6.13 Suitability of load-balancing policies for different regions of the irregular application space. 164

7.1 Histogram of thread granularities on the Cray T3D for the IC-Cedar application. 173

7.2 Histogram of thread granularities on the Cray T3D for the radiosity application. 173
7.3 Histogram of thread granularities on the Cray T3D for the Gröbner application. 174
7.4 Histogram of thread granularities on the Cray T3D for the phylogeny application. 174
7.5 The force calculation phase of IC-Cedar involves irregular spatial interactions (shown using arrows) to access fields of neighboring atom objects. 177
7.6 Overall performance impact of data locality and load balancing techniques for the IC-Cedar application. 180
7.7 Execution time breakdown showing impact of data locality techniques for the IC-Cedar application. 182
7.8 Execution time breakdown showing impact of load-balancing techniques for the IC-Cedar application. 184
7.9 Patches (circles) and interactions (boxes) in the hierarchical radiosity application. 185
7.10 Overall performance impact of data locality and load-balancing techniques for the radiosity application. 190
7.11 Individual impact of data locality techniques for the radiosity application. 192
7.12 Execution time breakdown showing impact of data locality techniques for the radiosity application. 193
7.13 Individual impact of load-balancing techniques for the radiosity application. 195
7.14 Execution time breakdown showing impact of load-balancing techniques for the radiosity application. 196
7.15 Polynomials, basis, and the pair queue in the Gröbner application. 198
7.16 Overall performance impact of data locality and load balancing techniques for the Gröbner application. 201
7.17 Execution time breakdown showing impact of data locality techniques for the Gröbner application. 203
7.18 Individual performance impact of load balancing techniques for the Gröbner application. . . 205
7.19 Execution time breakdown showing impact of load balancing techniques for the Gröbner application. 206
7.20 Tasks, solution nodes (circles), and the failure set (square) in the phylogeny application. . . 208
7.21 Overall performance impact of data locality and load-balancing techiques for the phylogeny application. 212
7.22 Execution time breakdown showing impact of data locality techniques for the phylogeny application. 214
7.23 Individual performance impact of load balancing techniques for the phylogeny application. . 215
7.24 Execution time breakdown showing impact of load balancing techniques for the phylogeny application. 216
7.25 Aggregate performance impact of run-time mechanisms across all applications. 223
7.26 Individual performance impact of data locality mechanisms across all applications. 224
7.27 Individual performance impact of load-balancing mechanisms across all applications. 225

CHAPTER 1

INTRODUCTION

Irregular and dynamic computations have seen widespread use in various applications such as molecular dynamics, particle simulations, and radiosity calculations that span a number of engineering and scientific domains. State-of-the-art techniques for these problems rely on sophisticated adaptive and hierarchical algorithmic cores (e.g., adaptive mesh refinement [1–3], and tree based n-body codes [4,5]) which reduce total computation by focusing effort where most beneficial. These algorithms, in turn, are conveniently expressed using pointer-based data structures such as linked lists, trees, and sparse matrices. The cumulative effect of constructing applications using these irregular algorithms and data structures can be a reduction in the required computation by one or more orders of magnitude. Although such reductions are impressive, they still fall considerably short of the performance expectations of scientists who wish to run their programs either for more time steps (in simulations), or with larger data sets than are possible on current-day workstations. Thus, there is considerable interest in coupling these algorithmic gains with the performance benefits realizable from parallel processing.

However, such dynamic computations are challenging to express and program efficiently on scalable parallel machines. Dynamic computations are characterized by the on-demand creation of irregular sized units of work (a function of the hierarchical and adaptive nature of the algorithms), and unpredictable data access patterns (a function of dynamic pointer-based data structures). These characteristics are poorly matched to the three predominant parallel programming models—data parallel, message passing, and shared memory. There are several reasons for this mismatch arising from distinct mechanisms for specifying concurrency, distribution, addressability, and synchronization in each of these programming models. Dynamic creation of work and data objects is incompatible with data parallel models which express concurrency in terms of independent operations on statically allocated data collections. Message passing requires the programmer to map the computation into a fixed number of address spaces and explicitly deal with the complexities of distribution, addressability, and synchronization using matching communication primitives. The shared memory paradigm alleviates some of these distribution and addressability problems by assuming a single shared address space; however, the programmer is still required to map the computation into a fixed number of threads which synchronize using hardware-supported word-level atomic operations. The complexity of building program synchronization structures on top of the word-level operations is further exacerbated by the use of pointer-based data structures.

On the other hand, high-level programming models based on dynamic thread creation and multithreading [6–13] enable convenient expression of irregular parallel computations. These models form the basis for several concurrent object-oriented languages [14–17] and message-driven systems [12]. Such models permit concurrency to be expressed in terms of arbitrary user-defined computation units (hereafter referred to as logical threads), which synchronize using object-level concurrency control. In addition, the models provide a global name space that simplifies management of data placement and distribution. The logical threads and a global name space enable natural expression of the program's concurrency structure and pointer-based data structures respectively. The underlying implementation is responsible for achieving efficiency by controlling the detailed mapping of the program to the underlying machine (exploiting program concurrency using physical threads and realizing the global name space), relieving the programmer from managing details of thread and data structuring.

Despite the above programmability advantages, such high-level programming models have not gained widespread use because they are challenging to implement efficiently on scalable parallel machines. High performance on such machines requires three elements:

1. *Local efficiency*, i.e., ensuring that a large fraction of the processor cycles are devoted to user computation

2. *Data locality*, i.e., minimizing costs to access shared data

3. *Load balance*, i.e., ensuring that all processors are cooperating to perform the user computation

However, realizing these performance elements is difficult due to challenges arising from properties of both the computational model and irregular applications. The model and applications combine to produce dynamic, irregular thread structures (e.g., granularity variations) and data access patterns. Thus, the main obstacle to widespread acceptance of parallel programming models based on dynamic multithreading is ensuring local efficiency, data locality, and load balance for fine-granularity threads and frequent communication operations in the presence of computation irregularity.

Although the above problem can be addressed entirely at either the compiler or the run-time system level, a good solution must include both components in complementary roles. Static compiler techniques have been successful in achieving local efficiency at the single thread level [18] but, in general, have been insufficient for managing global resources (e.g., processors, memories, and caches) because of the dynamic nature of the computation model. Thus, compiler techniques enable, but do not achieve, parallel efficiency. Consequently, in this research, we focus on developing *run-time techniques* for local efficiency, data locality, and load balance of dynamic multithreaded computations. Our techniques leverage compiler advances at the single thread level to further improve overall performance.

Our results show that dynamic multithreaded computations can achieve performance on scalable parallel machines, which is comparable to the best performance achievable via low-level programming approaches, indicating that high-level programming models need not sacrifice on performance. We show that there is no fundamental impediment to performance; however, the run-time techniques for thread management, communication, data locality, and load balance that contribute to overall performance must be designed differently from traditional implementations of these techniques. These run-time techniques must additionally adapt to the dynamic computation nature and provide performance robust over computation irregularity.

The rest of this chapter is organized as follows. Section 1.1 describes the thesis being validated in this research and briefly summarizes our approach. Section 1.2 presents an overview of the dissertation. The main contributions of the research are listed in Section 1.3. Finally, the dissertation organization is presented in Section 1.4.

1.1 Thesis

This research focuses on the development of run-time techniques for dynamic multithreaded computations which would allow their execution with the same efficiency as low-level approaches (such as C or Fortran with message passing) on scalable parallel machines. The thesis is best stated as follows:

> Dynamic multithreaded computations can be executed efficiently on scalable parallel machines using run-time local efficiency techniques which adapt to locality and thread distribution, and run-time data locality and load balance techniques which achieve performance robust over computation irregularity.

To validate this thesis, we describe the design and evaluation of a run-time execution framework for dynamic multithreaded computations. This framework developed in the context of the Illinois Concert System [19–22] addresses the overall efficiency problem by separating concerns of local and parallel efficiency.

Local efficiency is achieved by low-overhead thread management and communication techniques which deliver performance conditioned upon run-time locality and load balance. These mechanisms adapt to the dynamic nature of computation and serve as a robust building block for parallel efficiency. Parallel efficiency is achieved by data locality and load balance techniques which exploit knowledge of the application semantics and leverage properties of the target execution platform to provide performance robust over computation irregularity. Thus, our approach relies on parallel efficiency mechanisms to ensure good run-time locality and thread placement, which can then be recognized and exploited by the local efficiency mechanisms.

Individual techniques in the framework are evaluated using synthetic microkernels which allow convenient modeling of a range of application behaviors. The overall impact of the framework is evaluated using four large irregular applications.

1.2 Overview

This dissertation describes the design and evaluation of a run-time execution framework for dynamic multithreaded computations on scalable parallel machines. The following provides an overview of the dissertation, and corresponds to a chapter-by-chapter breakdown:

Background

In Chapter 2, we present relevant background on dynamic multithreading-execution models by describing a particular instance. We first introduce the concurrent object-oriented programming model and its execution model. We then provide a concrete context for this research, describing a specific concurrent object-oriented language (Illinois Concert C++), the target parallel platforms (Cray T3D and SGI

Cray Origin 2000), and an implementation platform for ICC++ on the target platforms (Illinois Concert System). We describe in some detail the capabilities of the Concert compiler and the performance of a baseline run-time system.

Problem Statement and Approach

In Chapter 3, we describe both the problems contributing to the current poor performance of dynamic multithreaded computations on scalable parallel platforms and our approach for addressing them. We first identify the performance challenges arising both due to the dynamic nature of the computation model and the characteristics of irregular applications. We then formally state the key subproblems that are the focus of this research. We motivate our approach by describing related work and its shortcomings. Next, we present an overview of our approach which separates concerns of local and parallel efficiency, using techniques for the former to provide a building block for the latter. The chapter also provides a classification of communication architectures with different capabilities — MSG, PUT/GET, LD/ST, and CC-LD/ST correspond respectively to architectures with support for basic send-receive messaging, one-sided remote memory access operations, non cache-coherent memory and atomic operations, and cache-coherent memory and atomic operations. This classification serves as a basis for the systematic evaluation of communication architecture support for run-time techniques in the rest of the dissertation.

Local Efficiency

In Chapter 4, we describe low-overhead thread management and communication mechanisms which ensure local efficiency conditioned upon run-time locality and load balance. These mechanisms adapt to the dynamic nature of computation, and serve as a robust building block for parallel efficiency. We first identify the requirements of an ideal local efficiency building block and demonstrate limitations of existing run-time techniques. We then present two techniques: *hybrid stack-heap execution* and *pull messaging*. Hybrid stack-heap execution is a thread-management technique which takes advantage of run-time locality to reduce overheads by coalescing computation into coarse-grained threads on the fly. Pull messaging is a robust communication mechanism which provides performance virtually insensitive to local processor activity and communication patterns. We evaluate the performance impact of these two mechanisms using a synthetic microkernel. Our results show that when applicable, each mechanism individually improves microkernel performance by up to 50%, with stack heap execution improving performance for applications with fine granularity threads, and pull messaging improving performance for applications that communicate frequently. Together, they increase significantly the range of application behaviors which can be efficiently supported.

Data Locality

In Chapter 5, we describe a latency-tolerant caching framework, *view caching*, for improving data locality of multithreaded computations. We first describe a microkernel evaluation of a state-of-the-art object-consistent distributed shared memory (DSM) implementation and find that it exhibits high sensitivity to processor unresponsiveness and unbalanced sharing on architectures such as MSG and PUT/GET, requiring a more supportive communication architecture with responsive, atomic operations against remote memory (e.g., LD/ST) for robust performance. Despite the availability of responsive operations in architectures such as PUT/GET, existing protocols are not able to exploit these characteristics for a performance advantage. These shortcomings motivate the view caching approach, which utilizes knowledge of the application data access-semantics to construct custom coherence protocols that require reduced synchronization and show increased tolerance to computation irregularity. Microkernel results show that view caching protocols reduce sensitivity to processor unresponsiveness and unbalanced sharing, even on less supportive architectures such as MSG and PUT/GET, despite higher per-operation costs. In addition, view caching protocols on a particular communication architecture achieve performance parity with object-consistent DSM protocols on a more supportive architecture, suggesting that customizing protocols using knowledge of application semantics can compensate for a lack of hardware support.

Load Balance

In Chapter 6, we describe a framework for load-balancing dynamic multithreaded computations. Our framework combines run-time object placement and thread placement heuristics to efficiently distribute threads with wide variations in granularity, while preserving data locality and application execution priorities. We first describe a simple model of dynamic task generation in a global object space and prove the intractability of two formulations of the load balancing problem. These results motivate a framework which organizes object and thread placement heuristics in a cost hierarchy; a compiler or a programmer selects the most efficient policy for each set of threads, which are then composed together to obtain load balance for the entire application. We then describe three components of the framework: language and run-time support to associate sets of threads with a particular load balancing policy and to compose multiple policies at run time, a library of object placement, thread placement and thread ordering policies, and a microkernel-based evaluation of the suitability of different policies for different regions of the irregular application space. The microkernel results show that no single policy provides best performance over the entire space; thus, each set of threads that behaves differently should be subject to a different load-balancing policy. Our results show that object placement mechanisms are suitable when applications exhibit low imbalance or small thread granularities. For higher thread granularity and higher degrees of imbalance, thread-placement policies are appropriate. Finally, when the application also requires prioritized thread execution, thread ordering mechanisms yield best overall performance.

Performance Evaluation

In Chapter 7, we present an application-based performance evaluation of the data locality and load-balancing mechanisms. Using four large irregular applications—IC-Cedar, Radiosity, Gröbner, and Phylogeny—we describe use of the view caching and load balancing frameworks, and evaluate the resulting performance. Our results demonstrate that these mechanisms are both individually effective and collectively sufficient, enabling each application to achieve performance on both the T3D and the Origin that is comparable to the best achievable via low-level programming means on large-scale parallel systems. Specifically, our mechanisms together yield an improvement in application performance of up to 10 times over existing data locality and load balancing techniques. In addition, the mechanisms enable performance on less supportive communication architectures such as MSG, PUT/GET, and LD/ST to approach that on the CC-LD/ST architecture, suggesting that the run-time mechanisms can compensate for a lack of hardware support. Individually, both view caching and load balancing frameworks contribute to performance: disabling either of them results in a speedup degradation of up to 10 times. View-caching mechanisms improve performance by reducing coherence time and idle time, both due to the use of asynchronous protocols and due to reductions in concurrency control delays because of overlap between detached read views and modifications to a shared object. Load-balancing mechanisms yield performance benefits arising from coherence time reductions due to locality-sensitive thread distribution, idle time reductions due to balanced thread distributions, and compute time reductions due to priority-sensitive thread distribution.

1.3 Contributions

The primary contribution of this dissertation is a collection of run-time techniques which enable high-level programming languages based on dynamic multithreading to achieve performance comparable to the best achievable via low-level programming means on large-scale parallel systems. Individual contributions are summarized below:

1. Run-time execution framework for dynamic multithreaded computations, which separates concerns of local and parallel efficiency, allowing independent development and optimization of techniques for each.

2. Local efficiency techniques for thread management and communication, which deliver performance conditioned upon run-time locality and load balance, and thereby significantly increase the range of application behaviors which can be efficiently supported:

   - An efficient thread management technique, *hybrid stack-heap execution*, which takes advantage of run-time locality situations to reduce overheads by coalescing computation into coarse-grained threads on the fly, resulting in thread creation costs as low as a C procedure call
   - A robust communication mechanism, *pull messaging*, which combines the advantages of distributed queuing and lazy receiver-initiated data transfer to provide performance which is virtually insensitive to output contention and unbalanced sharing

   These techniques differ from comparable mechanisms in literature [9, 23–27] in their explicit design for robustness and ability to exploit different run-time locality and load balance situations.

3. Parallel efficiency techniques for data locality and load balance, which exploit knowledge of application semantics and leverage properties of the target execution platform to provide performance robust over computation irregularity:

   - A latency-tolerant custom coherence framework, *view caching*, which utilizes knowledge of application data-access semantics to build protocols that require less synchronization and, hence, are more tolerant to unresponsive processors. The primary differences of view caching from other custom protocol approaches [28–30] are its ability to express use-site specific optimizations and leverage various communication architectures.

- A load-balancing framework that achieves overall application load balance by composing individual policies for each set of threads exhibiting different behaviors, permitting thread distribution to be simultaneously low-overhead and sensitive to data locality and thread execution priority. While the component policies are not new, what is novel is the use of multiple policies within the same application and the mechanisms used for their composition.

4. Demonstration of individual effectiveness and collective sufficiency of the run-time techniques, which shows that irregular applications expressed in a high-level dynamic multithreaded computation model can achieve performance comparable to the best achieved using low-level approaches on scalable parallel machines, and indicates that programs expressed in a high-level language need not sacrifice on efficiency.

5. Systematic evaluation of the qualitative impact of various communication architectures on the design and performance of data locality and load balancing mechanisms, which (a) shows that the mere presence of responsive operations in an architecture such as PUT/GET is no guarantee of good performance since existing data locality and load-balancing protocols are unable to exploit these characteristics for performance, and (b) makes a case for the inclusion of hardware support for remote memory access because it enables high-performance messaging and software shared memory implementations that are robust to computation irregularity.

1.4 Organization

The rest of this dissertation is organized as follows. Chapter 2 provides relevant background for dynamic multithreading computations, first describing general programming and execution models, and then presenting a concrete language, target platform and implementation context for this research. In Chapter 3 we formally state the problem and summarize our approach. Chapter 4 describes techniques for local efficiency which serve as the foundation for parallel efficiency mechanisms described in Chapters 5 and 6 respectively. An application based evaluation of the parallel efficiency mechanisms is described in Chapter 7. Finally, Chapter 8 summarizes the results, discusses their significance, and identifies avenues for future work. For completeness, Appendices A and B present raw execution data of the various microkernels and applications respectively.

CHAPTER 2

BACKGROUND

This chapter provides relevant background on the dynamic multithreading execution model by describing its instantiation in a particular programming model. The chapter consists of five sections. Sections 2.1 and 2.2 introduce the concurrent object-oriented programming and the dynamic multithreaded execution models, respectively. Sections 2.3, 2.4, and 2.5 provide the concrete context for this thesis. In Section 2.3, we present a specific concurrent object-oriented programming language, Illinois Concert C++ (ICC++). Section 2.4 describes the target scalable parallel platforms. Finally, in Section 2.5, we describe the Illinois Concert System, which comprises an optimizing compiler and run-time system for executing ICC++ programs on the above platforms.

2.1 Concurrent Object-Oriented Programming

In this research, we study a high-level programming model which supports the programming of dynamic, irregular parallel applications. This model, concurrent object-oriented programming, brings the benefits of object-oriented programming to bear on the challenges of concurrency and distribution management. For sequential programs, the encapsulation provided by object-oriented techniques supports code reuse and separate design (modularity), enabling the construction of larger, more complex systems more rapidly than previously possible. Concurrent object-oriented languages support program modularity through object encapsulation, allowing the hiding of decisions about concurrency, distribution and other irrelevant details.

In this section, we first describe how concurrency and synchronization are expressed in the model, and then discuss its high-level programming features.

2.1.1 Concurrency and synchronization

The concurrent object-oriented programming model expresses computation as interactions among a set of autonomous communicating entities (objects). Objects reside in a globally shared name space and communicate via method invocations. In general, the concurrent object-oriented model extends the sequential model with concurrent statements, concurrent objects, and multiaccess object collections.

Concurrent statements specify non-binding (or elective) concurrency by specifying a partial order among program statements (typically, method invocations). Each method invocation corresponds to a logical thread of execution. Invocations need not block the caller thread: Synchronization between caller and callee threads is achieved via futures [31].

Concurrent objects are abstract data types which provide a consistent interface in a concurrent environment. Object-level concurrency control is enforced by selective processing of method invocations.

Multiaccess object collections are object groups which are indexable using a single name. The single name allows a single-object abstraction to be replaced by a multiaccess abstraction (where concurrent operations can scale with the number of group objects), without changing its interface for the rest of the program.

This general programming model forms the basis for several object-based parallel languages [12,15, 32,33]. In Section 2.3, we describe a specific language, Illinois Concert C++ (ICC++), in more detail. A theoretical basis for reasoning about the concurrent object-oriented model can be found in the work on Communicating Sequential Processes (CSP) [34] and Actors [35].

2.1.2 High-level programming features

The concurrent object-oriented model provides three high-level features which simplify the expression of dynamic, irregular applications.

Object-oriented programming improves organization and modularity of programs, reducing the complexity of large parallel programs. Object-level concurrency control encapsulates concurrency decisions and allows the programmer to reason about atomic operations at the level of user-specified operations. Similarly, multi-access object collections naturally encapsulate decisions about object distribution.

Global object name space allows all data objects to be accessed uniformly, separating data and task placement choices from functional specification. In addition, freeing the programmer from explicitly managing different names for local and remote objects simplifies considerably the construction of sophisticated data structures required for irregular applications.

Elective concurrency allows the programmer to identify all concurrent operations in the program (including those created as part of the execution), leaving it to the underlying system to adapt the exploited concurrency to available parallelism. This flexibility permits a natural, non-binding expression of the concurrency and synchronization structure of dynamic, irregular applications.

These high-level features simplify considerably the task of expressing irregular applications as compared to either the data-parallel or the message-passing model, two popular alternatives. Data parallel models have limited applicability because they require concurrency to be cast as independent operations on distinct data elements. On the other hand, the message-passing model exemplified by the single-program multiple-data (SPMD) style requires that the computation be mapped into a fixed number of concurrent threads communicating via matching communication operations.

However, the improved programmability comes at a cost: concurrent object-oriented programming models require a more sophisticated execution model. This execution model is described in the next section.

2.2 Dynamic Multithreaded Execution

In this section, we describe an execution model for concurrent object-oriented programs. Computations are executed by mapping the communicating objects across the machine, and interleaving processor resources among dynamically created concurrent activities (hence the name, dynamic multithreaded execution). We first describe the entities that are required (at run time) to realize the computation expressed in the program. We then discuss the characteristics of target hardware platforms that are the focus of this research, and conclude the section by describing how the execution entities are mapped to these platforms.

2.2.1 Execution entities

Three kinds of execution entities are required to realize the computation expressed in the program: objects, threads, and scheduling and distribution structures. Figure 2.1 shows these entities, described in more detail below.

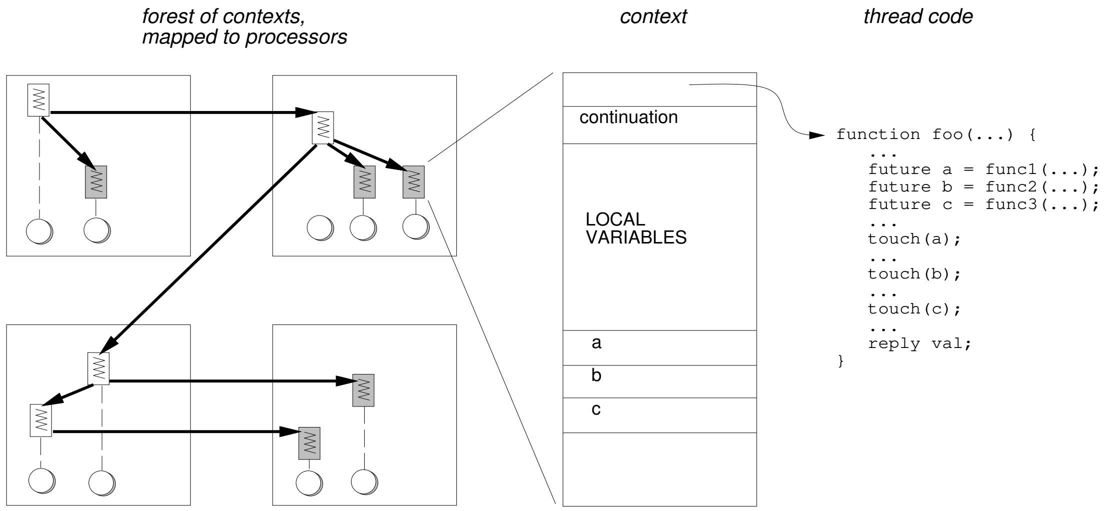

Figure 2.1 Execution model showing processing nodes (large boxes), contexts (small boxes), objects (circles), and thread creation (dark arrows). A context includes space for the thread local variables, future locations for pending requests, and the continuation (linkage to the requester). The shaded contexts are active (or ready); on each processor, these contexts are linked into a scheduling structure (not shown).

Objects represent instances of the data abstractions expressed in the program. At run time, they are represented as memory regions laid out with fields for instance variables and concurrency control. The instance variable fields are used for method computation, while concurrency control fields are used for selective processing of method invocations. Objects are the units of data distribution.

Threads represent exploited concurrent operations, adapted by a compiler from the logical concurrency identified by the programmer. Thread creation involves allocating a context, depositing the thread arguments into it, and initiating execution of the thread code. Since thread creation need not suspend the caller, dynamic thread creation results in a forest of active contexts. The context provides storage for the local variables of the thread (similar to a procedure stack frame), and fields for future-based thread synchronization. Threads are the units of scheduling.

Scheduling and distribution structures manage the mapping of objects and threads across the processors of the machine. On each processor, scheduling structures multiplex the processor resources among the active threads.

2.2.2 Characteristics of target platforms

While it is possible to specify an execution model for concurrent object-oriented programs on arbitrary multiple-instruction multiple-data (MIMD) parallel machines, in this thesis we restrict our attention to a subset which is based on the following assumptions about the basic structure of future parallel machines.

Processing nodes. We assume commodity processing nodes based on workstation or PC-like architectures which support fast local processing with a deep memory hierarchy (two or more levels). This structure dictates that data reuse in the registers and upper levels of the memory hierarchy is essential for performance. Our focus is on developing techniques applicable for commodity hardware platforms, so we specifically exclude processing nodes with hardware support for fine-grained interleaving of threads (e.g., [36–38]).

Communication network. We assume a scalable, low-latency, high-bandwidth communication network with inexpensive access (i.e., the network can be accessed without operating system calls).

Processor-network interface. Our parallel machine model encompasses a variety of communication architectures for the processor-network interface. Specifically, we consider three kinds of communication architectures:

1. *Two-sided*: A send-receive interface (e.g., the CM-5 [39]) where processor participation on both the sender and the receiver is required for the completion of a communication operation,

2. *One-sided*: A remote memory access interface (e.g., the Cray T3D [40]) where a processor can directly access memory on a remote node, but such accesses are not cache-coherent, and

3. *Zero-sided*: An interface supporting cache-coherent loads and stores (e.g., the SGI Cray Origin 2000 [41]) where data transfer between two memories can be achieved without requiring the explicit participation of either processor.

While the specific semantics of the communication architecture is flexible, we assume that access to some memory in the system is significantly faster than accessing other memory (e.g., there is a local memory). This implies that locality of access is important. This notion of “locality domains” is compatible with a broad range of existing and announced scalable parallel machines [40–44].

Synchronization support. To ensure that the techniques described in this work are applicable on a large set of machines, we make no assumptions about special mechanisms for local or global synchronization.

Within this machine class, the fundamental performance issues are exploiting data locality within and between nodes, and achieving a balanced work distribution throughout the machine.

2.2.3 Mapping entities to hardware resources

Mapping the execution entities to hardware resources involves two tasks:

1. Distributing objects across processing nodes and synthesizing a global object name space
2. Scheduling threads both across and within processing nodes

Object distribution across the processors of the machine is controlled by the distribution structures, which can implement specific policies for the allocation of individual objects and distribution of object collections. Location-independent access is supported by referencing objects using global names. Because the target platform does not always support a global name space directly in hardware, the underlying system must synthesize it in software by detecting accesses to remote data objects and translating them into communication operations.

Threads execute, by default, on the processor where the target object (of the corresponding method) resides. More sophisticated scheduling policies can be implemented using the thread scheduling structures. Multiple threads mapped to the same processing node are managed by a node scheduler which interleaves the processing resources among ready threads. A thread suspends when it touches the value of one of its futures, or when it needs to wait due to object-level concurrency control. In such cases, the scheduler activates the next ready thread. A blocked thread becomes ready once the future it is blocked on has been filled by another thread holding the continuation (the right to fill the future). The execution model does not specify any particular order for granting resources of a processing node to ready threads, other than to require that the scheduler support fairness guarantees specified by the programming model.

2.3 Illinois Concert C++ (ICC++)

In this section we describe a specific concurrent object-oriented language, Illinois Concert C++ (ICC++), used to express the benchmarks and applications described in this thesis. ICC++ is an extension of the sequential programming language C++ [45], augmenting the basic object-oriented programming system with concurrent statements, concurrent objects, and multiaccess object collections. We first describe the basic programming interface of ICC++ (Section 2.3.1), and its performance optimization interfaces (Section 2.3.2). The latter represent programming language support for the data locality and load balance mechanisms described in Chapters 5 and 6, respectively. We then describe the expression of an example application program in ICC++ (Section 2.3.3) to demonstrate use of the language constructs and to highlight the programmability advantages of the concurrent object-oriented model. The reader is referred to [17, 32, 46] for a complete language definition.

2.3.1 Basic programming interface

Concurrent statements. ICC++ programs declare concurrency by annotating standard blocks (i.e., compound statements) and loops with the `conc` keyword as illustrated below:

```
conc {                   // statement
  Foo *b = new Foo(a); // s1
  r1 = b.meth1();      // s2
  r2 = b.meth2();      // s3
  r3 = r1 + r2;        // s4
}
```

```
conc while (i < 5) {   // s5
  a->foo(i);           // s6
  i = i + 1;           // s7
}
```

A `conc` block defines a *partial* order amongst its statements, allowing concurrency while preserving local data dependences. Any pair of statements in a `conc` block may execute concurrently unless an identifier appearing in both is assigned or declared in one of them. For instance, in the block example (left), statements `s2` and `s3` can execute concurrently. In the loop example (right), although the loop unfolds sequentially (due to dependence between statements `s5` and `s7`), the execution of statement `s6` across different iterations can be concurrent. Thus, rules for the `conc` annotation preserve local dependencies, enabling easy construction of concurrent programs by incrementally refining sequential ones. Also, both forms of `conc` indicate *nonbinding* concurrency, allowing programmers to specify concurrency and depend on the underlying system to manage computation granularity.

Concurrent objects. ICC++ extends the C++ object model to provide synchronization by defining a simple semantics for object concurrency—method invocations on an object execute as if they had exclusive access. The state of the object is sequentially consistent; however, operations across multiple objects may not be. This raises the level of consistency from single memory locations to user-defined objects. Note that synchronization does not require any syntactic additions to the language.

ICC++ object semantics also define concurrency guarantees to enable reasoning about deadlock freedom. Two independent invocations are guaranteed to be concurrent if neither one can possibly write any object state that might be read or written by the other.

Multiaccess object collections. ICC++ provides object collections to support large-scale concurrency. ICC++ uses the C++ array syntax to define an indexable set of element objects and a separate collection state. Unlike array elements, the constituent elements of a collection are aware of the collection, allowing collections to implement an aggregate behavior and interface. ICC++ provides predefined methods to allow element objects to query about the size of the collection and their place in it, as well as for the collection object to access the individual elements of the collection. Thus, object collections provide a convenient place for expressing parallelism and data distribution.

ICC++ collection syntax is demonstrated in the example below. The declaration creates two classes: `Accum` and `Accum[]` representing the elements and the whole collection respectively. The `Accum[]` interface provides a consistent data abstraction, while internal calls to `Accum::add` and `Accum::total` may be concurrent. Thus, this concurrent accumulator can be substituted for a sequential one:

```
class Accum[] {
  int Accum::local;

  int Accum::add(int a) { local += a; }
  int Accum::total(void) { return Accum[]::this->total(); }
  int Accum[]::total(void) {
    int total;
    conc for (i=0; i < size(); i++)
      total += (*this)[i].local;
    return total;
  }
}
```

2.3.2 Performance-optimization interfaces

In addition to the basic programming interfaces described above, ICC++ also contains three kinds of implementation-specific performance optimization interfaces for managing data locality and load balance. These interfaces control data placement, data consistency, and thread scheduling, and represent the language support required by the data locality and load-balance mechanisms described in Chapters 5 and 6, respectively. A description of these interfaces has been included here for completeness.

Data placement. Data placement and collocation for individual objects can be specified using the standard C++ placement syntax to overload the `new` operator [45]; an implementation-specific function `location_of` is assumed available that provides the run-time location of its argument. Data placement and alignment for object groups can also be managed through object collections—the user can specify either standard layouts (e.g., block, cyclic) or custom distributions at creation time. These various kinds of placement control are illustrated in the code fragment below:

```
Foo *obj1 = new ( LOCAL ) Foo(4);                // object placement
Foo *obj2 = new ( location_of(obj1) ) Foo(5);    // object collocation
Accum *coll1 = new ( BLOCK ) Accum[16];          // collection distribution
```

Data consistency. The data-consistency interface allows the language implementation to use relaxed consistency models for higher performance. This interface is expressed using a `with_local` annotation to blocks which specifies a set of objects that need to be localized and provides information about their access semantics. In the following code fragment, the annotation declares that access to objects `obj1`, and `obj2` within the block is read-only, allowing the system to acquire them using efficient protocols.

```
function( obj1, obj2, ... ) {
  with_local( obj1, READ_ONLY, obj2, READ_ONLY ) {
    /* use obj1 and obj2 here */
    ... = func( obj1, obj2, ... );
  }
  ...
}
```

Thread scheduling. The thread scheduling interface provides control over thread-placement and thread-execution ordering. The thread-placement interface controls data locality and load balance from the alternative perspective of threads rather than objects. The thread-execution interface is useful in applications based on search which prioritize promising tasks for higher efficiency. Both controls are specified by annotating the thread creation site (i.e., the method invocation statement) using the `!scheduler` keyword. The argument following the keyword specifies the use of an implementation-specific custom scheduler for scheduling the thread. In addition, thread execution ordering requires that its priority be specified using the `!priority` keyword. In the following code fragment, the thread corresponding to `thread_function` executes with priority `arg1` and is managed by a global priority scheduler (a custom scheduler).

```
...
( thread_function( arg1, arg2, ... )
    !priority arg1 !scheduler GLOBAL_PRIORITY );
...
```

2.3.3 Example: An ICC++ program

In this section, we discuss the expression of an example irregular application, hierarchical radiosity. A detailed discussion of this application can be found in Chapter 7. The intention here is to illustrate a concrete use of ICC++ syntax, and to highlight the programmability advantages of the concurrent object-oriented model.

The radiosity method computes the global illumination in a scene containing diffusely reflecting surfaces by expressing the radiosity of an elemental surface patch as a weighted linear function of the radiosities of all other patches. We base our implementation upon a hierarchical algorithm due to Hanrahan et al. [47] which starts with the initial patches comprising the scene and computes light transport between pairs of patches, hierarchically subdividing each patch as needed to ensure accuracy. Each patch maintains interaction lists of potentially visible neighbors. Computation proceeds in iterations, where, for each patch, we compute its radiosity due to all patches on its interaction list, subdividing it or the other patches hierarchically as required. Subdivided patches acquire their own interaction lists and contribute a weighted term to their parent's radiosity. The visibility calculation dominates the computation time. Figure 2.2 pictorially shows the refinement of patches and the interaction lists.

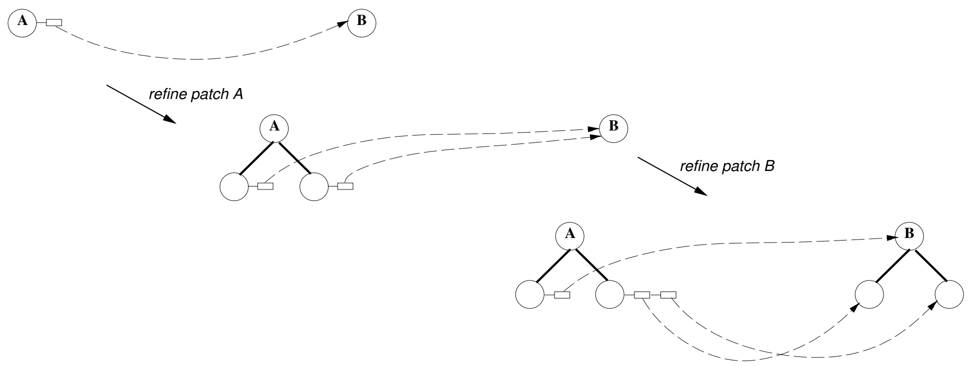

Figure 2.2 Patches (circles) and interactions (boxes) in the hierarchical radiosity application. Patches become refined on demand as the computation progresses.

Though it is relatively straightforward to identify concurrency in the algorithm—parallelism can be exploited across all input patches, across child patches of a subdivided patch, and across neighbor patches stored in the interaction list—the concurrency structure is very irregular. The application is irregular in how deep each input patch can be refined, in how many interactions need to be computed for each patch, and even how long a particular interaction calculation takes. In other words, both thread granularity and thread creation structures are irregular. Consequently, it is difficult to capture the above parallelism in a message-passing or data-parallel program without significantly altering the sequential algorithm.

On the other hand, the high-level nature of the concurrent object-oriented model, and the simple ICC++ extensions allow the parallel code to be constructed incrementally from the sequential version. Figure 2.3 shows a portion of the sequential C++ version of the application code. Figure 2.4 shows the corresponding parallel ICC++ code: the differences arising from the use of ICC++ keywords are highlighted using boxes. We discuss, in turn, the sequential C++ expression, the parallel ICC++ expression, and use of high-level features of the concurrent object-oriented model.

Sequential C++ code. Figure 2.3 shows the structure definitions of interactions (`Inter`), and patches (`Patch`), and representative function bodies showing the code executed in each iteration. The `main` function iterates until global convergence, calling the `process_rays` method on each input patch during an iteration. The `process_rays` method in turn first iterates over the patch's interaction list, analyzing the error to optionally refine either the current patch or the destination patch. Refinement can introduce new interactions for which visibility needs to be calculated. Once a patch and its interactions have been refined as required, the radiosity for the patch is computed as a weighted sum of its interactions and contributions from its children.

Parallel ICC++ code. Figure 2.4 shows the code changes that were required to convert the sequential version into a parallel code.

Concurrency is specified using the `conc` annotation for loops and blocks. The loop level annotation is used in three places: the `process_rays` invocation in the `main` function, and the `error_analysis` and `compute_visibility` invocations in the `process_rays` method. The annotation at the block level is used to specify that the `process_rays` invocations on the patch children *can* execute concurrently. These annotations capture the various levels of parallelism in the application: across all input patches, across the child patches, and across all interactions associated with a patch. Note that the semantics

```
struct RGB {
  float   r, g, b;

  RGB&    operator +( RGB& );
  RGB&    operator *( float );
};

struct Inter {
  Patch   *dest;
  float   ff, vis;
  Inter   *next;
};

struct Patch {
  RGB     rad;
  /* other fields */
  ...
  /* parent and child patches */
  Patch   *parent;
  Patch   *center, *top;
  Patch   *left, *right;
  /* resolved interactions */
  Inter   *inter;
  /* unresolved interactions */
  Inter   *vis_undef;

  void error_analysis( Inter * );
  void compute_visibility( Inter * );
  RGB process_rays( void );
};

void
Patch::compute_visibility( Inter *cur ) {
  Patch *dest = cur->dest;
  cur->vis =
    func( cur->ff, /* fields */,
          /* dest-> fields */ );
}

void
main( void ) {
  RGB total_rad;
  ...
  while ( !converged ) {
    /* over top level patches */
    for ( i=0; i<num_top; i++ )
      total_rad +=
        top_level[i]->process_rays( );
  }
  ...
}
```

```
RGB
Patch::process_rays( void ) {
  Inter *cur;
  RGB   rcenter, rtop, rleft, rright;
  int   flag = FALSE;

  do {
    /* iterate over resolved inters */
    cur = inter; inter = NULL;
    for ( ; cur; cur = cur->next ) {
      /* optionally refine patches */
      analyze_error( cur );
    }

    /* check if any unresolved inters */
    if ( vis_undef == NULL ) flag = TRUE;
    else {
      /* for all unresolved inters */
      cur = vis_undef; vis_undef = NULL;
      for ( ; cur; cur = cur->next )
        compute_visibility( cur );
    }
  } while ( !flag );

  rad = 0; cur = inter;
  /* accumulate patch contributions */
  for ( ; cur; cur = cur->next ) {
    rad += ( cur->dest->rad *
             (cur->ff*cur->vis) );
  }
  /* add contributions from children */
  if ( center != NULL )
    {
      rcenter += center->process_rays();
      rtop += top->process_rays();
      rleft += left->process_rays();
      rright += right->process_rays();
    }
  rad += 0.25 * ( rcenter + rtop +
                  rleft + rright );

  return rad;
}
```

Figure 2.3 C++ expression of the hierarchical radiosity application.

```
struct RGB {
  float   r, g, b;

  RGB&    operator +( RGB& );
  RGB&    operator *( float );
};

struct Inter {
  Patch   *dest;
  float   ff, vis;
  Inter   *next;
};

struct Patch {
  RGB     rad;
  /* other fields */
  ...
  /* parent and child patches */
  Patch   *parent;
  Patch   *center, *top;
  Patch   *left, *right;
  /* resolved interactions */
  Inter   *inter;
  /* unresolved interactions */
  Inter   *vis_undef;

  void error_analysis( Inter * );
  void compute_visibility( Inter * );
  RGB process_rays( void );
};

void
Patch::compute_visibility( Inter *cur ) {
  Patch *dest = cur->dest;
  with_local ( this, READ_ONLY,
               dest, READ_ONLY ) {
    cur->vis =
      func( cur->ff, /* fields */,
            /* dest-> fields */ );
  }
}

void
main( void ) {
  RGB total_rad;
  ...
  while ( !converged ) {
    /* over top level patches */
    conc for ( i=0; i<num_top; i++ )
      total_rad +=
        top_level[i]->process_rays( );
  }
  ...
}
```

```
RGB
Patch::process_rays( void ) {
  Inter *cur;
  RGB   rcenter, rtop, rleft, rright;
  int   flag = FALSE;

  do {
    /* iterate over resolved inters */
    cur = inter; inter = NULL;
    conc for ( ; cur; cur = cur->next ) {
      /* optionally refine patches */
      analyze_error( cur );
    }

    /* check if any unresolved inters */
    if ( vis_undef == NULL ) flag = TRUE;
    else {
      /* for all unresolved inters */
      cur = vis_undef; vis_undef = NULL;
      conc for ( ; cur; cur = cur->next )
        ( compute_visibility( cur )
          !scheduler WORK_SHARING );
    }
  } while ( !flag );

  rad = 0; cur = inter;
  /* accumulate patch contributions */
  conc for ( ; cur; cur = cur->next ) {
    rad += ( cur->dest->rad *
             (cur->ff*cur->vis) );
  }
  /* add contributions from children */
  if ( center != NULL )
    conc {
      rcenter += center->process_rays();
      rtop += top->process_rays();
      rleft += left->process_rays();
      rright += right->process_rays();
    }
  rad += 0.25 * ( rcenter + rtop +
                  rleft + rright );

  return rad;
}
```

Figure 2.4 ICC++ expression of the hierarchical radiosity application.

of the `conc` annotation ensure that local dependencies are respected, permitting the parallel version to retain the original sequential C++ code structure.

Synchronization is naturally expressed using ICC++'s object concurrency semantics. Access to fields of a patch object in the `compute_visibility` method can proceed concurrently with the patch's `process_rays` invocation. Note that ICC++'s semantics, which give each method apparent exclusive access to object state implies that no additional source code is required to maintain patch state consistency.

The performance optimization interfaces are used at two places. The data-consistency interface is used in the `compute_visibility` method to inform the underlying implementation of the read-only use semantics for the patch objects pointed to by the `this` and `dest` pointers. The underlying system uses this information to cache a view of the patch object with low overhead. In the `process_rays` method, the concurrent invocations to `compute_visibility` illustrate use of the thread placement interface. In this case, the annotation specifies that `compute_visibility` invocations are managed by a work sharing scheduler which ensures load balance.

Thus, the simple concurrency, synchronization, and performance optimization syntactic interfaces in ICC++ permit the construction of the parallel version by incrementally modifying the original sequential code. In addition, the orthogonal nature of the performance optimization interfaces permits a separation of data and task placement choices from functional specification.

High-level programming model. Two high-level features of the concurrent object-oriented model assist in simplifying parallel expression: the global object name space, and elective concurrency.

A global-object name space permits both local and potentially remote objects to be accessed uniformly. Such uniform access is essential to avoid additional code in the parallel version which explicitly generates communication code to access remote objects. For instance, in the `compute_visibility` method, both the patch objects referred to by `this` and `dest` pointers are accessed uniformly, even though one or both of them may be remote to where the thread executes. Consequently, the parallel version of the `compute_visibility` method can retain essentially the same structure as the corresponding method in the sequential version.

The elective concurrency semantics permit the programmer to simply specify all concurrency in the application, leaving it up to the underlying implementation to make granularity decisions. This feature is particularly significant given the irregular, data-dependent nature of the application; requiring the programmer to construct balanced, coarse-grained threads would significantly complicate the parallel expression.

Summary. The parallel expression of irregular applications is simplified considerably using a high-level concurrent object-oriented model which provides a global name space and elective concurrency. In addition, simple concurrency, synchronization, and performance optimization extensions in the ICC++ language permit the parallel code to be incrementally constructed from the sequential version. However, this ease of programming comes at the cost of requiring a more sophisticated implementation.

2.4 Hardware Platforms

This research uses two hardware platforms—the Cray T3D [40] and the SGI Cray Origin 2000 [41]—which possess the characteristics of the generic target architecture defined in Section 2.2.2. The choice of these two platforms was motivated mainly by the desire to evaluate the run-time mechanisms developed in this research on a range of communication architectures, specifically the *two-sided*, *one-sided*, and *zero-sided* architectures discussed earlier. To avoid cross-platform performance comparisons, we emulate the communication architectures on each platform whenever necessary: the T3D allows emulation of two-sided and one-sided architectures, while the Origin 2000 allows emulation of two-sided, one-sided, and zero-sided architectures.

The performance attributes of these platforms (obtained from [48–50]) are summarized in Table 2.1,[1] and their architectural features are described in more detail below. The following discussion is restricted to only features relevant for this thesis.

Table 2.1 Performance summary of hardware platforms.

PLATFORM	PROCESSOR	ACCESS LATENCY (in ns (cycles))				NETWORK BANDWIDTH (in MB/s)
		L1	L2	local	remote	
Cray T3D	150 MHz DEC 21064	6.7 (1)	- -	145.0 (22)	850.0 (128)	160
SGI Cray Origin 2000	195 MHz MIPS R10000	5.1 (1)	56.4 (11)	310.0 (60)	945.0 (185)	1250

[1] The bandwidth numbers presented here refer to the sustainable bandwidth from the network into the node card, which for the purposes of this research, is the limiting factor.

2.4.1 Cray T3D

The Cray T3D is a physically distributed memory machine with support for a shared address space, consisting of up to 512 nodes interconnected with a scalable three-dimensional torus network [51]. Each processing node consists of a 150 MHz DEC 21064 [52] processor (with an 8 KB direct-mapped L1 cache, and no L2 cache) and up to 64 MB of memory, extended with a shell of Cray custom circuitry, called the DTB Annex. The DTB Annex allows a processor to read or write all of the machine's memory without the assistance of the remote processors. The T3D processing element structure is shown in Figure 2.5. The addressing of remote memory is not the same as local memory; it requires special operations in the DTB Annex. The T3D nodes include additional hardware for block transfers and message queues, but because of the associated overheads, that hardware is relevant only for large messages.

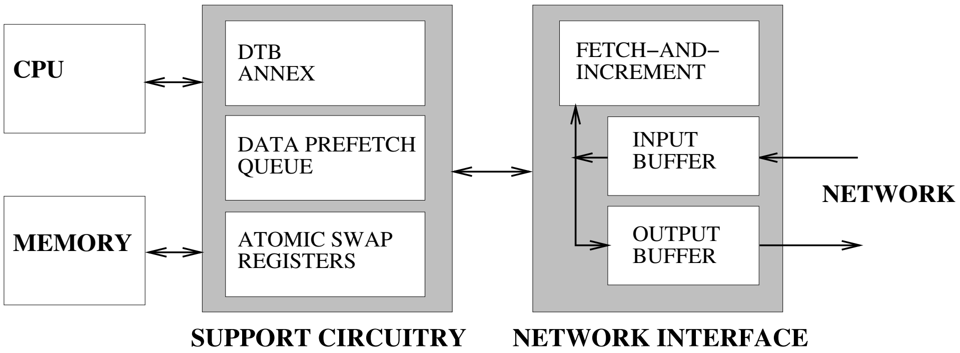

Figure 2.5 The T3D processing element.

T3D nodes also support three special memory operations for synchronization and latency hiding: fetch-and-increment, atomic swap, and prefetch [53]. First, each processor has a fetch-and-increment register that can be read or written by other processors. Reading the fetch-and-increment register atomically increments the register's contents and returns the original contents to the requesting processor. Second, each processor has a special register, called the *swaperand*, which is used to implement atomic swaps with remote memory locations. To swap, a program first sets up a remote memory location as a swap location then reads that location, causing the current value to be returned while atomically swapping the value of the swaperand register into the remote location. Third, processors can prefetch data; each prefetch returns a word to the 16-word prefetch queue. This is most useful for masking remote memory latency. These special operations all involve off-chip access, and thus are expensive (greater than 10 processor cycles).

Each node of the Cray T3D runs a microkernel based upon the Cray UNICOS operating system [54]. The machine is operated in a space-shared fashion with each job getting a dedicated view of the machine.

Thus, the architecture of the T3D has the following implications for performance. The small size of the L1 cache, its direct-mapped nature, and the absence of an L2 cache imply that sequential performance is very sensitive to cache-line alignment. In particular, for our dynamic multithreading model, user computation is likely to see some degradation of performance over the corresponding sequential code, due to its interleaving with code implementing the run-time mechanisms. Furthermore, the hardware-supported remote memory access primitives and atomic operations provide a very flexible interface for implementing the run-time communication and object-caching mechanisms.

2.4.2 SGI Cray Origin 2000

The SGI Cray Origin 2000 is a cache-coherent nonuniform memory access multiprocessor (CC-NUMA), consisting of up to 512 nodes interconnected by a scalable bristled hypercube network based on the SGI Spider router chip [55]. Although memory modules are physically distributed (being part of the processing node), the Origin provides global addressability, employing a distributed shared memory (DSM) architecture with cache-coherence maintained via a directory-based protocol. Figure 2.6 shows relevant portions of the Origin processing element structure. Each node consists of one or two 195 MHz MIPS R10000 [56] processors (with a 32 KB two-way set associative L1 cache, and a 4 MB three-way set associative L2 cache) and up to 4 GB of coherent memory (along with its corresponding directory memory). The Hub chip encapsulates the cache-coherence logic, as well as the processor and network interface logic [50, 57]. It provides data paths between the processors and the memory, as well as between the memory and the network.

The cache coherence protocol works with 128-byte cache lines, and is similar to the Stanford DASH invalidation-based protocol [58]. Enhancements include request forwarding to reduce the latency of interprocessor communication, a clean-exclusive state to minimize latency on read-modify-write operations, and cache dropping of clean-exclusive or shared data without notifying the directory to reduce use of memory/directory bandwidth. Origin supports a full bit-vector directory up to 64 nodes (128 processors), and beyond that chooses dynamically between a full bit vector and a coarse bit vector [59] scheme. The processor interface contains logic for implementing the request tracking for both processors, as well

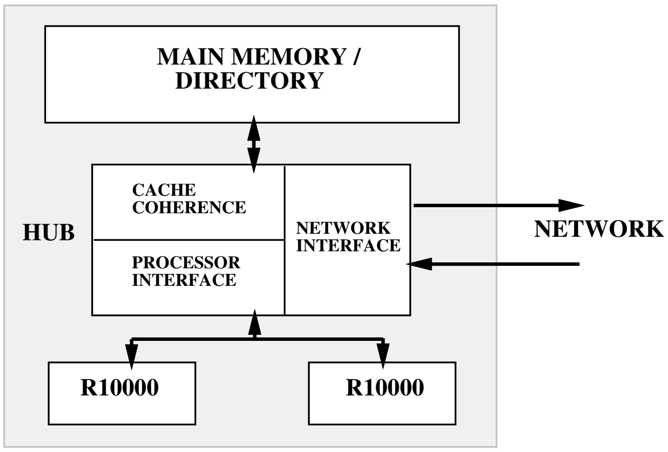

Figure 2.6 The Origin 2000 processing element.

as for generating interrupts to the processors. The network interface takes messages to and from the network and other parts of the Hub and is responsible for generating multiple unicast invalidation messages. In addition, the Hub chip also provides support for uncached fetch-and-op operations against memory locations. These operations complement the standard MIPS load-linked/store-conditional (LL/SC) instructions on the R10000 by reducing serialization for highly-contended locks and barrier operations.

The Origin 2000 runs a version of Cellular IRIX, which multitasks the entire machine among several parallel jobs. The operating system provides support for gang-scheduling processes of a job (based on resource availability), although these processes may be restarted on another processor after a context switch to improve throughput. In addition, the operating system supports automatic page migration to improve the performance of an individual process.

Thus, the architecture of the Origin 2000 has the following implications for performance. The large size of the L1 and L2 caches and their set-associative nature implies that, unlike the T3D, sequential performance is likely to remain robust, even when computation portions of the code are interleaved with code implementing the various run-time mechanisms. The internode architecture implies that communication and object-caching mechanisms need to be implemented on top of cache-coherent loads and stores, potentially resulting in unnecessary memory traffic. Finally, the combination of LL/SC instructions and hardware supported fetch-and-op operations provides implementations with a flexible set of low-overhead synchronization primitives.

2.5 The Illinois Concert System

This research was conducted in the context of the Illinois Concert System [19–22], a project of the Concurrent Systems Architecture Group at the University of Illinois. The Concert system provides a high-performance implementation platform for concurrent object-oriented languages on scalable parallel platforms. It currently supports two languages: Concurrent Aggregates [15] and Illinois Concert C++ [32] (the latter is described in more detail in Section 2.3). The system consists of an optimizing compiler and a high-performance run-time system that includes the mechanisms described in this thesis. In addition, it facilitates program development by including an emulator, a symbolic debugger (ParaSight [60]), and tools for performance monitoring [61]. The Concert system is available for public use, along with accompanying documentation and reference programs from the following web site: `http://www-csag.cs.uiuc.edu`.

The Concert system has been a vehicle for extensive research on compiler optimization and run-time techniques over the past five years [18, 62–71]. These techniques represent a range of aggressive optimizations that have been used to demonstrate high performance in absolute terms on a wide range of applications [21, 70, 72, 73]. In effect, the Concert system automatically addresses many of the concerns which programmers must manage in lower level programming models, such as the following:

- Procedure granularity and virtual function call overhead
- Compound object structuring
- Thread granularity
- Distributed name management
- Initial data placement

The rest of this section is organized as follows: In Sections 2.5.1 and 2.5.2, we briefly describe the structure and capabilities of the Concert compiler and run-time system, respectively. In Section 2.5.3, we discuss in greater detail base Concert run-time mechanisms for thread management and communication. Their performance provides a competitive baseline for comparing the performance advantages of techniques described in the rest of the thesis.

2.5.1 The Concert compiler

The Concert compiler employs aggressive static global program analyses and transformations, exploiting information about the high-level semantics of the concurrent object-oriented programming model, to produce code which, on uniprocessor platforms, executes with efficiency comparable to C programs [18,66]. The discussion here provides only an overview of the compiler structure and its state of the art optimization capabilities. The interested reader is referred to the web site listed above and the following papers and theses for additional details [18,62–69].

The global program analyses [18, 65] use a unified flow analysis framework to obtain a variety of information: types of variables to resolve dynamic dispatches prevalent in the object-oriented model, relative locality of objects (if specified at the language level), and container object relationships used in storage optimizations. The analysis is *context sensitive*, adapting in a demand-driven manner to the program structure. To prevent information loss, the analysis creates contexts (representing program environments) for differing uses of classes (for example, polymorphic containers), and methods (for example, differing types for a given argument at given call sites). These contexts are created on demand when the analysis needs to distinguish some property, leading to an adaptive flow analysis technique that yields precise information (to the extent required) without incurring exponential complexity.

The information gathered from the above analyses is used to perform four interprocedural static optimizations [66,68] that reduce the overheads associated with the concurrent object oriented programming model:

- *Method inlining* eliminates method invocation overhead, and has significant impact due to the tendency in object oriented programming styles towards small method bodies.
- *Procedure cloning* creates specialized, optimized versions of method bodies based upon the calling context.
- *Object inlining* allocates objects within other objects, reducing object creation and access costs, storage management overheads, and improving cache locality.
- *Access region expansion* expands the dynamic extent of *access regions*—program regions with guaranteed locality and access control properties. These regions are used by the Concert compiler to generate code, which is speculatively optimized based upon run-time guards. Access region

expansion reduces run-time checking overhead, and additionally creates larger basic blocks for scalar optimizations.

These aggressive analyses and transformations enable the Concert compiler to successfully eliminate overheads arising from the high level object-oriented nature of the programming model. However, as we shall see in Section 3.1, its ability to produce large enough thread grain sizes for efficiency is restricted by lack of static information about the dynamic multithreaded computation model. Consequently, in the Concert system, although the compiler and the run-time system work in close coordination, they focus on different aspects of implementation efficiency. The compiler addresses intrathread overheads by generating sequentially efficient thread code and by controlling the amount of state that needs to be saved and restored across context switches. In addition, it generates code which takes advantage of fortuitous object locality and thread scheduling at run time. Thus, the compiler enables, but cannot achieve, efficient execution. On the other hand, the run-time system provides efficient thread and communication primitives, as well as mechanisms which ensure appropriate object locality and thread scheduling for load balance.

2.5.2 The Concert run-time system

The Concert run-time system provides primitives for communication and thread management, as well as data-locality and load-balancing mechanisms. These mechanisms achieve performance an order of magnitude higher than do corresponding primitives in vendor-supplied communication and thread libraries. Additionally, performance remains robust over the unpredictable dynamic behavior characteristic of irregular applications. Efficient primitives and a close coupling with the compiler permits the run-time system to avoid programmer management of thread granularity, distributed name management, and initial data placement.

The run-time system has the same underlying structure on both distributed memory and cache-coherent shared memory parallel platforms. The unifying notion is that of an *address space*, within which (potentially) multiple *workers* interact using hardware-supported atomic operations and cache-coherent loads and stores against memory. Interaction across address spaces is via software communication and object caching mechanisms. An address space for distributed memory platforms comprises a processor and its private memory. On the other hand, on a shared memory platform, there are a spectrum of choices ranging from one processor per address space to a global address space which includes all processors.

The run-time system for ICC++ consists of language-independent and language-dependent portions. The language-independent portion consists of four core modules: communication, thread management, memory management, and object caching. The communication module supports the Fast Messages interface [74]. The thread management module provides support for thread scheduling, quiescence detection, and custom user schedulers (see Chapter 6). The quiescence detection support employs a message-efficient algorithm by Sinha, Kalé, and Ramkumar [75]. The memory-management module supports allocation of memory blocks in a single address space, as well as aligned stripes across address spaces. In addition, it implements a distributed garbage-collection algorithm, based on a generalization of the Baker's Treadmill Algorithm [76], to reclaim unreferenced memory. The object caching module supports software distributed object memory based on object consistency [77–79] and view caching protocols (see Chapter 5). All modules use a hierarchical design that reduces contention when used with multiple processors per address space.

The language-dependent portion implements the execution model described in Section 2.2, providing support for objects, contexts, object-level concurrency control, futures and continuations. In addition, it provides an interface to the compiler for creating object collections and invoking methods on objects, as required by the concurrent object-oriented programming model described in Section 2.1.

These components of the run-time system are described in more detail in the rest of this thesis. Section 2.5.3 describes the basic communication and thread management mechanisms. In Chapter 4, we describe enhancements to these basic mechanisms that deliver robust performance for irregular applications. In Chapter 5, we discuss object caching, and in Chapter 6, object placement and thread scheduling for load balance.

2.5.3 Base run-time system mechanisms

In this section, we describe basic Concert run-time mechanisms for thread management and communication. These primitives represent a high-performance implementation of typical operations found in standard communication and thread libraries [80, 81], and provide a baseline for evaluating the performance advantages of techniques described in the rest of the thesis. The primitives incur an order of magnitude lower cost compared to corresponding vendor-supplied implementations (see Tables 2.2 and

2.3), mainly due to the use of a specialized compiler which enforces a usage discipline. The compiler allows the run-time system to provide more efficient primitives, albeit with restricted functionality.

Thread management. The thread management mechanisms provide primitives for creation and deletion, scheduling, and synchronization. We describe in turn, their interface, implementation, and performance on our target platforms.

Interface. Thread creation and deletion utilize the `thread_create` and `thread_delete` primitives. The `thread_create` operation allocates a context (the thread's activation frame), deposits the supplied arguments into it, and initiates execution of the thread code (pointed to by `function_ptr`) relative to this newly allocated context. The `thread_delete` operation reclaims the context for the thread `t`:

```
Thread* thread_create(function_ptr, args, ...);
void    thread_delete(Thread* t);
```

Thread scheduling is achieved using two primitives: `thread_enqueue` attaches the thread to the node scheduler, and `thread_dequeue` selects the next ready thread and initiates execution:

```
void thread_enqueue(Thread* t);
void thread_dequeue();
```

Thread synchronization is achieved via the future mechanism. Primitives are provided to make, resolve and touch futures. The primitive `make_future` sets up a context location as a future and returns a continuation. The primitive `touch_future` checks the future, returning TRUE if it has been resolved, else FALSE. The latter is used by the calling thread to suspend on the future. The primitive `resolve_future` stores a value into the future and enqueues the blocked thread, if any:

```
Continuation make_future(return_location);
int          touch_future(return_location);
void         resolve_future(Continuation contin, value);
```

Implementation. In our base implementation, a context is a heap-allocated structure whose allocation and deletion operations are performed efficiently using a free list. Use of a heap-allocated context is enabled by a specialized compiler that generates code for saving and restoring thread state (into the context) at suspension points. The compiler can also optimize thread switching cost by managing live state at suspension points.

The thread scheduler maintains a list of ready threads as a singly-linked list. The `thread_enqueue` primitive adds the context to the tail of the list. The primitive `thread_dequeue` dequeues the head of the list and initiates execution of the thread code via an indirect function call. More sophisticated thread-scheduling policies described in Chapter 6 (e.g., priority-based queuing) can be built on top of this scheme.

Thread synchronization is implemented inexpensively using a context location to store the future (essentially a distinguished bit string). The primitive `make_future` tags the location as a future and returns a continuation which contains a pointer to the location. The primitive `touch_future` just involves a tag check. The primitive `resolve_future` first performs a tag check to see if a thread has blocked on the future, and then stores a value into it, enqueuing any blocked threads. In addition, the Concert run-time system provides support for counting futures as well as a general `touch` primitive permitting synchronization on multiple futures, amortizing the cost of thread restart.

Performance. Table 2.2 summarizes the costs of the above primitives on the Cray T3D and the SGI Cray Origin 2000. The costs on the Origin are based on a run-time system configuration of one processor per address space; thus, they do not account for overheads arising from lock contention in configurations with multiple workers per address space. However, in the latter case, the mechanisms are implemented in a hierarchical fashion (with processor-private structures and a shared global structure), essentially making the contention negligible.

Table 2.2 Costs of basic Concert thread management primitives on the T3D and the Origin. Costs of comparable operations in the Nexus multithreading system [82] on the IBM SP-2 [44] are shown for comparison.

<table>
<tr><th colspan="2" rowspan="3">OPERATION</th><th colspan="4">CONCERT</th><th colspan="2">NEXUS</th></tr>
<tr><th colspan="2">T3D</th><th colspan="2">ORIGIN</th><th colspan="2">SP-2</th></tr>
<tr><th>cycles</th><th>μs</th><th>cycles</th><th>μs</th><th>cycles</th><th>μs</th></tr>
<tr><td rowspan="2">Creation</td><td>`thread_create`</td><td>150</td><td>1.00</td><td>37</td><td>0.19</td><td>2144</td><td>32.0</td></tr>
<tr><td>`thread_delete`</td><td>100</td><td>0.67</td><td>24</td><td>0.12</td><td></td><td></td></tr>
<tr><td rowspan="2">Scheduling</td><td>`thread_enqueue`</td><td>40</td><td>0.27</td><td>18</td><td>0.09</td><td rowspan="7">3350</td><td rowspan="7">50.0[a]</td></tr>
<tr><td>`thread_dequeue`</td><td>40</td><td>0.27</td><td>18</td><td>0.09</td></tr>
<tr><td rowspan="5">Synchronization</td><td>`make_future`</td><td>20</td><td>0.14</td><td>8</td><td>0.04</td></tr>
<tr><td>`touch_future`</td><td>11</td><td>0.07</td><td>8</td><td>0.04</td></tr>
<tr><td>(if empty)</td><td>22</td><td>0.14</td><td>14</td><td>0.07</td></tr>
<tr><td>`resolve_future`</td><td>10</td><td>0.07</td><td>12</td><td>0.06</td></tr>
<tr><td>(if waiting)</td><td>50</td><td>0.35</td><td>31</td><td>0.16</td></tr>
</table>

[a] The 50.0μs number is the additional cost incurred in Nexus for thread-to-thread communication as opposed to handler-to-handler communication. It includes the cost of synchronizing using condition variables and switching between two Nexus threads and in functionality is comparable to the scheduling and synchronization of a Concert thread.

Heap-allocated contexts enable thread creation, switching, and synchronization deletion costs of less than 1μs on the T3D and less than 0.2μs on the Origin, an order of magnitude cheaper than corresponding primitives in multithreading systems such as Chant [6] and Nexus [13]. These latter systems require a dedicated stack per thread because they rely on a sequential compiler infrastructure that does not provide any support for saving and restoring thread state across suspensions.

Communication. Communication operations arise whenever a thread accesses a remote data object or interacts with a remote thread. We describe in turn the interface, implementation, and performance of Concert communication primitives.

Interface. The Concert communication primitives use a subset of the low-level Fast Messages [74, 83] Version 1.0 interface. The subset consists of two send primitives and one receive primitive:

```
typedef void function_ptr(...);
void    send_4(int rnode, function_ptr *fptr, arg1, arg2, arg3, arg4);
void    send(int rnode, function_ptr *fptr, void *buf, int size);
int     extract();
```

Each message send is associated with execution of a handler at the destination node (similar to Active Messages [23]). The `send_4` is optimized for transferring a small number of register arguments (up to 4) to the destination node, while the `send` primitive is more general, accepting an arbitrary sized buffer and the message length as arguments. Message reception requires use of the `extract` primitive which executes the handlers for all pending messages and returns TRUE if any messages were processed.

The advantage of the lean interface is that it minimizes buffer management in the messaging layer, permitting an efficient implementation. However, as with Active Messages, this interface assumes a certain usage discipline. Specifically, one must ensure that protocols involving sends from within message handlers must be deadlock free. This is another place where the availability of a compiler helps reduce the cost of primitive run-time mechanisms: a compiler can enforce the required discipline.

Implementation. Implementations of the above interface take full advantage of the machine capabilities on the Cray T3D and the SGI Cray Origin 2000. These implementations represent traditional techniques for implementing low-overhead messaging layers on message-passing and shared-memory architectures respectively, and are described in more detail below.

The T3D implementation of the Fast Messages interface [84] preallocates a set of message buffers on each processor. Message sends require the source processor to reserve one of the buffers at the destination, and transfer data into it. Message extracts require the destination processor to detect filled buffers and invoke the corresponding handler. Our implementation makes use of hardware support for fetch-and-increment and remote memory access [53] to perform buffer management and data transfer without involving the destination processor. This decouples the sending processor from destination processor activity, improving communication performance. The fetch-and-increment register is used to index the preallocated buffers. The source node obtains a valid index by performing a fetch-and-increment operation with respect to the destination node, then transfers data using remote stores. Message completion at the receiver is detected by a tag at the end of the message buffer.

The Origin implementation of the Fast Messages interface is built around the notion of address spaces discussed earlier in Section 2.5.2. Each address space has a *communication endpoint* which can be serviced by any of the workers in the address space. Unlike the per-processor private message buffer pools in the T3D implementation, the Origin implementation uses a common preallocated pool of message buffers in the cache-coherent address space. Message sends require the source processor to allocate one of the above buffers, fill it with the message data, and link the buffer onto a queue associated with the destination endpoint. Message extracts require the destination processor to delink the buffer, and invoke the corresponding handler. Our implementation utilizes hardware support for uncached fetch-and-op operations [50] to reduce the locking overhead associated with enqueuing a buffer into the destination queue.[2] To reduce secondary cache misses associated with storing data into and retrieving data from message buffers, our implementation splits up the logically shared buffer pool into per-processor pools; a processor deallocates a recently read buffer into its local pool, ensuring cache reuse when it allocates a buffer to send to someone else. The communication endpoint is protected against conflicting accesses using low-overhead lock primitives built from the LL/SC instruction pair.

Performance. Table 2.3 summarizes the costs of implementing the above interface on the Cray T3D and the SGI Cray Origin 2000.

The Concert communication primitives exploit a lean messaging interface to achieve a latency of only 10% to 20% of the latency of vendor communication libraries. The overhead numbers reveal an interesting aspect of the two implementations. In the T3D implementation, a message send completes only after the data has been "pushed" to the destination. On the other hand, in the Origin implementation, when the send completes, the data still resides in the source processor's cache; it is only when

[2] An alternative queuing implementation can use LL/SC instructions to atomically swap the new buffer with the contents of the queue tail pointer at the destination; however, on the Origin, a LL/SC instruction pair incurs inordinately large overheads when the target memory location is shared by more than processor.

Table 2.3 Latency and software overheads of Concert communication primitives on the T3D and the Origin for various message sizes. The vendor communication library latency numbers are shown for comparison.

PLATFORM	OPERATION (all costs in μs)		MESSAGE SIZE (in bytes) 16	32	64	128	256
T3D	Overhead	send	1.84	1.85	2.03	2.71	3.53
		extract	0.73	0.73	0.73	0.73	0.73
	One-way Latency		6.07	6.19	6.37	7.04	7.91
	Vendor Library (PVM) Latency		28.04	28.37	29.65	31.94	36.92
Origin	Overhead	send	0.57	0.65	0.75	0.95	2.37
		extract	2.64	2.78	2.83	3.28	5.83
	One-way Latency		5.15	5.25	5.39	7.00	10.62
	Vendor Library (MPI) Latency		15.13	15.45	14.93	17.42	19.56

the destination processor executes an extract that the data moves into its cache. Consequently, the send overheads dominate in the T3D implementation, whereas the extract overheads are dominant for the Origin implementation.[3] Further, the Concert primitives overlap communication and computation; consequently, the non-overhead portion of latency can be effectively eliminated.

Summary. Concert base primitives for thread management and communication utilize the existence of a specialized compiler to incur an order of magnitude less cost than corresponding mechanisms in portable libraries and multithreaded run-time systems. In addition, the compiler can customize the bundling of the run-time mechanisms affecting procedure call boundary crossings (e.g., the compiler can inline some of the run-time primitive calls), further reducing cost in situations where it has static information about thread interactions [85].

However, as we shall see in Chapter 3, the presence of these low-overhead primitives is not sufficient to overcome the performance challenges presented by the irregular nature In Chapter 4, we describe how the implementation of these primitives can be enhanced to achieve decoupled high local (intranode) efficiency that is conditioned upon good locality and load balance.

[3] The send overhead for 256-byte messages on the Origin is high because of costs associated with segmenting the message into multiple internal message buffers.

2.6 Summary

In this chapter we have provided relevant background for this thesis, describing the concurrent object-oriented programming and dynamic multithreaded execution models, and their instantiation in the context of a specific language (ICC++), set of target machines (Cray T3D and SGI Cray Origin 2000), and an implementation platform (the Illinois Concert system).

Concurrent object-oriented programming brings the benefits of object-orientation to bear on challenges of concurrency and distribution management. Computations are expressed as interactions among objects which reside in a globally shared name space. The programming model extends sequential object-oriented models with concurrent statements, concurrent objects, and multiaccess object collections. Its high-level features—object-oriented programming, a global object name space, and elective concurrency—simplify the expression of irregular parallel applications.

Dynamic multithreading executes concurrent object-oriented programs by mapping communicating objects across the machine and interleaving processor resources among concurrent activities (threads). The execution model defines the mapping of objects and threads to a generic parallel machine consisting of commodity processing nodes, low-latency and high-bandwidth communication networks, and diverse processor-network interfaces.

ICC++ is a concurrent object-oriented language which extends C++ with concurrency annotations, object-level concurrency control, and multiaccess object collections. Simple syntactic extensions and high-level programming features allow parallel programs to be constructed by incrementally modifying existing sequential codes.

The Cray T3D and the SGI Cray Origin 2000 serve as the target platforms for this research. Both machines are built around commodity processing cores, provide user-level access to a low-latency and high-bandwidth communication network, and allow emulation of various communication architectures. On both platforms, the fundamental performance issues are exploiting data locality within and between nodes, and achieving a balanced work distribution throughout the machine.

The Illinois Concert system is a high-performance implementation platform for concurrent object-oriented languages such as ICC++ on scalable parallel platforms. The Concert compiler employs aggressive static global program analyses and transformations, exploiting information about the high-level semantics of the concurrent object-oriented programming model, to produce code which, on uniprocessor platforms, executes with efficiency comparable to C programs. The Concert run-time system provides mechanisms for communication, thread management, data locality, and load balance, and serves as the implementation context for the run-time techniques described in the thesis. The thread management and communication primitives achieve an order of magnitude better performance than corresponding primitives in vendor-supplied libraries.

CHAPTER 3

PROBLEM AND APPROACH

In this chapter, we describe the problems which contribute to poor performance of dynamic multithreaded computations on scalable parallel platforms, and our approach for addressing them. The chapter is organized into four sections. Section 3.1 identifies the key performance challenges arising both due to the dynamic nature of the computation model and the characteristics of irregular applications. In Section 3.2, we formally state the subproblems that are the focus of this research. Section 3.3 motivates this work by describing how related work inadequately addresses these problems. In Section 3.4 we present an overview of our approach, and describe an organizing structure for the rest of the thesis. Finally, Section 3.5 summarizes the key points of the chapter.

3.1 Performance Challenges

In this section, we describe the challenges in achieving high performance for dynamic multithreaded computations on scalable parallel platforms. We discuss, in turn, the elements of performance, and challenges arising from the dynamic nature of the computation model and from characteristics of irregular applications.

3.1.1 Elements of performance

High performance on scalable parallel platforms of the kinds described in Section 2.2.2, requires the following three elements:

1. *Local efficiency*, i.e., ensuring that a large fraction of the processor cycles are expended doing user computation

2. *Data locality*, i.e., minimizing costs (in the form of communication operations) to access shared data objects

3. *Load balance*, i.e., ensuring that all processors are cooperating in performing the user computation

In our computation model, the first element requires that the thread code make effective use of architectural features such as registers and caches, and standard resource usage conventions such as stacks. In addition, the user computation must be able to amortize the overheads of communication and thread operations required by the model. The second element reduces the communication overheads required to realize the shared object space, and additionally affects the fraction of time that a processor spends waiting for a response to its communication requests. The third element ensures that a processor is not idle while there is work in the system, ensuring greater throughput.

In the predominant parallel computing models, namely data-parallel, message-passing, and shared-memory, these elements are achieved by a combination of static compiler analysis and user specification. For example, in data-parallel programs, the restricted nature of the model, as well as the regular computation structure, enables a compiler to make good decisions about how to partition the code to obtain local efficiency, data locality, and load balance. In message-passing programs, the onus is on the user to achieve these objectives. Shared-memory programs rely upon the user to achieve local efficiency and load-balance, leaving it up to the hardware to ensure data locality.

However, achieving these elements in the context of dynamic multithreaded computations requires a different approach that must meet the following challenges posed both by the model and the characteristics of irregular applications that benefit from it.

3.1.2 Dynamic computation models

The dynamic multithreading model is characterized by dynamic creation of objects and threads, and demand-driven run-time interleaving of resources among threads mapped to a particular processor. The dynamic nature limits the extent to which static compiler analysis or even profiling can predict patterns of resource usage. This problem persists despite the state-of-the-art-compiler analyses described in Section 2.5.1.

The limitations of static analysis for dynamic parallelism and dynamic data structures preclude two kinds of optimizations that are feasible in data-parallel models:

- *Communication optimization:* The compiler cannot statically partition data objects to minimize or aggregate communication.

- *Granularity optimization:* The compiler cannot coalesce logical threads into coarser-granularity physical threads which can amortize thread management overheads.

Thus, limitations of static analysis have two implications:

1. The implementation must achieve local efficiency, data locality, and load balance using *run-time* mechanisms

2. These mechanisms must incur low overhead to efficiently support the fine-granularity threads and resulting frequent communication operations.

3.1.3 Characteristics of irregular applications

In addition to the challenges presented by the dynamic nature of the multithreading model, performance is further affected by the nature of applications of interest. As discussed earlier, our goal is to support applications that exhibit irregularity along several dimensions—data, control, and concurrency—because these applications require the high-level features of the concurrent object-oriented model. Such applications demonstrate the following characteristics which affect performance:

- *Irregular thread structure:* Because threads are created dynamically based on ongoing computation to focus effort where beneficial, thread structure exhibits three forms of irregularity. First, irregular applications exhibit an uneven distribution of work across the processors. Second, these threads exhibit wide variations in granularity. And third, in some search-based symbolic applications, thread execution must be prioritized to minimize redundant work [86].

- *Unpredictable data access:* Because shared object access is intertwined with the computation and may evolve as a result of it, irregular applications exhibit unpredictable and unbalanced patterns of shared data object access.

The above characteristics have the following key implication: not only do applications require dynamic mechanisms for data locality and load balance which improve performance by adapting to the irregular nature of the computation, the implementation of these mechanisms must be robust over computation irregularity.

Specifically, data locality and load balancing mechanisms must deal with both *spatial* and *temporal* irregularity situations which arise due to the interleaving of processor resources inherent in the dynamic multithreading model. Figure 3.1 depicts the two situations by showing the execution timelines of two processors with interleaved thread execution and communication operations.

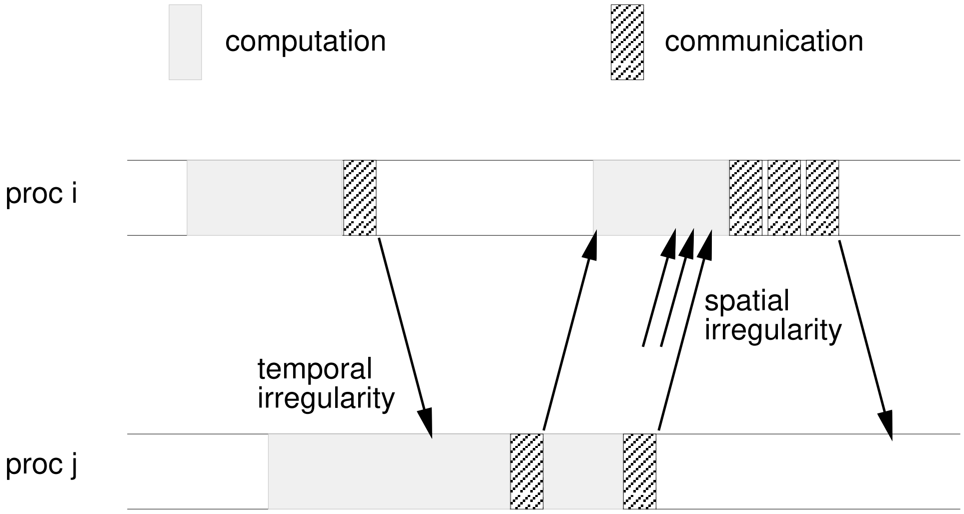

Figure 3.1 Spatial and temporal irregularities in implementations of dynamic multithreaded computations. The figure shows execution timelines of two processors (proc i, proc j).

Spatial irregularity situations arise from unbalanced thread and data access distribution across the processors, resulting in some processors receiving more communication requests than others. Temporal irregularity situations arise when communication operations are unsynchronized with computation across the processors, resulting in increased latencies and output contention.

3.2 Problem Statement

The performance challenges described in the previous section can be divided into three subproblems, all in the context of the target execution platforms described in Section 2.2.2.

1. How to design efficient run-time mechanisms for *local efficiency*, specifically, communication and thread management, that achieve good performance even with fine-granularity threads and frequent communication operations.

2. How to design efficient run-time mechanisms for *data locality* that adapt to evolving computation and deliver performance robust over computation irregularity.

3. How to design efficient run-time mechanisms for *load balance* that can be used to distribute threads with wide granularity variations, while preserving data locality and necessary execution priorities.

This research addresses each of these three subproblems in turn. Section 3.3 discusses shortcomings in related work to motivate the need for new solutions. Our solutions for each of these three subproblems are described in Chapters 4, 5, and 6 respectively.

3.3 Previous Work

In this section, we briefly discuss previous related efforts and the key problems that remain. Detailed descriptions of related work, from the perspective of individual run-time techniques, can be found in Sections 4.2.3, 4.3.3, 5.7, 6.7, and 8.2.

3.3.1 Thread management and communication

Many researchers have studied parallel programming approaches which rely on interleaving processor resources among multiple communicating threads of control. All such systems provide support for thread management and communication and can be broadly classified into two groups: coarse-grained multithreaded systems, and fine-grained multithreaded systems.

Coarse-grained multithreaded systems, which include standard communication [80, 87] and thread libraries [81], unified run-time systems [6, 82, 88, 89], or complete programming systems [10, 14, 90–92] all suffer from the drawback that their thread overheads are high, in the range of tens of microseconds, limiting their applicability to only coarse-grained irregular applications. With few exceptions [88], communication operations are poorly integrated with ongoing computation and incur high overheads for frequent, small-sized communication operations.

Fine-grained multithreaded systems, particularly those which implement concurrent object-oriented languages [16, 27, 93, 94] have focused more on language expression issues with less complete compilers and few large-scale experiments. More efficient implementations [7–9, 12, 33, 95, 96] contain low-overhead communication and thread mechanisms. However, these techniques are tied to specific language contexts (e.g., run to completion threads in functional [7] and message-driven [12, 95] languages, and static thread scheduling in dataflow languages [8]) which prevents their applicability in our language context. Additionally, the techniques do not yield good performance in the face of computation irregularity.

3.3.2 Data locality

Two popular approaches for run-time management of data locality in irregular applications are separable communication and distributed shared memory.

Separable communication approaches, as exemplified by the inspector-executor technique [97–99], separate out and independently optimize the computation and communication phases. While suitable for applications where the data access pattern in a phase is independent of ongoing computation, they cannot be used in the context of irregular applications where the data access patterns evolves with the computation.

Distributed shared memory (DSM) approaches provide a shared address space on distributed memory computers without hardware support for cache-coherence. Both page-based [11, 100–102] and object-based approaches [77–79, 103–108], even those relying on weak consistency models [109], either incur high overhead or achieve poor performance in the presence of computation irregularity. Systems supporting protocol customization [28, 29, 107, 110, 111] suffer from the drawback that they are not prescriptive, providing little insight into how to optimize protocols for performance in the presence of irregularity. Recently, researchers [102, 112, 113] have examined the use of additional hardware support, such as the communication architectures described in Section 2.2.2 to mitigate the problems of pure software DSM approaches. However, most of these efforts have focused on page-based systems. Little is known if the same optimizations will improve performance in the context of irregular fine-grained sharing, and if the benefits persist with a less supportive set of hardware mechanisms.

3.3.3 Load balance

Our load balancing focus differs from the unrelated problem of task allocation in distributed systems [114–116], where the objective is to distribute load to minimize turnaround times for job execution. Related work can be organized into two major categories: medium-grained and fine-grained approaches.

Medium-grained approaches require programmers to explicitly manage placement of threads [6, 13, 117], which freezes the load balancing structure into the program, or incur high overheads due to exchanges of load information between processors [10, 118–120], which makes them unsuitable for fine-grained computations.

Fine-grained approaches have predominantly controlled thread placement by controlling the location of data objects and having threads execute local to data that they access [9, 19, 27], which yields poor performance if threads access objects in an unbalanced fashion. Randomized thread scheduling approaches [7, 24] control thread placement directly and are theoretically optimal [121, 122], but these approaches suffer from the practical drawback that they are locality oblivious.

3.3.4 Summary

Previous approaches for thread and communication management, data locality, and load balance do not achieve good performance on dynamic multithreaded computations. Most mechanisms incur high overheads, making them unsuitable for use with threads that exhibit wide variations in granularity and require frequent communication operations. The few mechanisms that incur low overhead are unable to deliver high performance in the face of computation irregularity which arises from unbalanced thread creation, granularity variations, need for prioritized execution, and unpredictable and unbalanced data access patterns. These shortcomings motivate our approach, which is described in the next section.

3.4 Solution Approach

In this section, we provide an overview of our approach and describe the structure for the rest of the thesis.

Previous approaches suffer from the drawback that their mechanisms incur high overheads and yield poor performance in the face of irregularity. Our approach focuses on the design of efficient mechanisms for local and parallel efficiency which yield robust performance. It consists of two components:

- *Thread management and communication mechanisms for local efficiency*, which yield performance that is conditioned upon run-time locality and load balance. These mechanisms provide a robust building block for the parallel efficiency mechanisms.

- *Data locality and load balance mechanisms for parallel efficiency*, which exploit application-level information and properties of the target execution platform to provide performance robust over computation irregularity.

These two components reflect the fact that local efficiency for irregular applications on our target execution platforms is not orthogonal to the issues of data locality and load balance. The latter determine the set of active threads on a processor, as well as run-time locality of data objects, in turn affecting communication and thread operations, hence, local efficiency. Our approach relies on the parallel efficiency mechanisms to ensure good run-time locality and thread placement, which can then be recognized and exploited by the local efficiency mechanisms. Thus, local efficiency mechanisms need to yield robust performance; their robustness permits independent design of the parallel efficiency mechanisms.

The above two components are described in more detail below.

3.4.1 Local efficiency

Local efficiency mechanisms yield good thread and communication performance conditioned upon run-time locality and load balance. In order to serve as an effective building block for parallel efficiency, these mechanisms must be designed to

1. exploit run-time locality and thread placement to improve performance, and
2. decouple activity on different processing nodes, minimizing the disruption on the local activity of a processing node by other processors.

In Chapter 4, we describe one mechanism of each kind—*hybrid stack-heap execution* exploits run-time locality situations to yield efficient thread management, and *pull messaging* provides robust communication performance decoupled from local processing node activity. Chapter 4 also includes an evaluation of the performance of these mechanisms on our two target platforms, the Cray T3D and the SGI Cray Origin 2000.

3.4.2 Parallel efficiency

Parallel efficiency mechanisms improve data locality and load balance, building upon the local efficiency mechanisms. The distinguishing features of our mechanisms is that they are both low overhead and achieve good performance despite computation irregularity. These goals are met by mechanisms which exploit

1. knowledge of application data access and thread structure semantics, and
2. specific communication architecture capabilities of the underlying target platform.

Chapters 5 and 6 describe data locality and load balancing mechanisms (*view caching* and *custom load balancing framework*), respectively, which satisfy the above goals. The performance of these mechanisms is evaluated using four irregular applications in Chapter 7.

Classification of communication architectures. The parallel efficiency mechanisms are designed and evaluated in the context of a common set of four architectural interfaces which are representative of several recent and announced scalable parallel architectures. These interfaces systematically cover the space of two-sided, one-sided, and zero-sided communication architectures identified in Section 2.2.2, and are described in more detail below.

We consider a generic node architecture consisting of a processor and memory unit connected to the interconnection network via a network interface (NI) unit.[1] The four interfaces differ in the semantics of communication operations available, and data paths between the processor, the memory unit, and the network interface. Figure 3.2 pictorially shows these interfaces in the context of moving data from the source processor's memory (left) to the destination processor's memory (right):

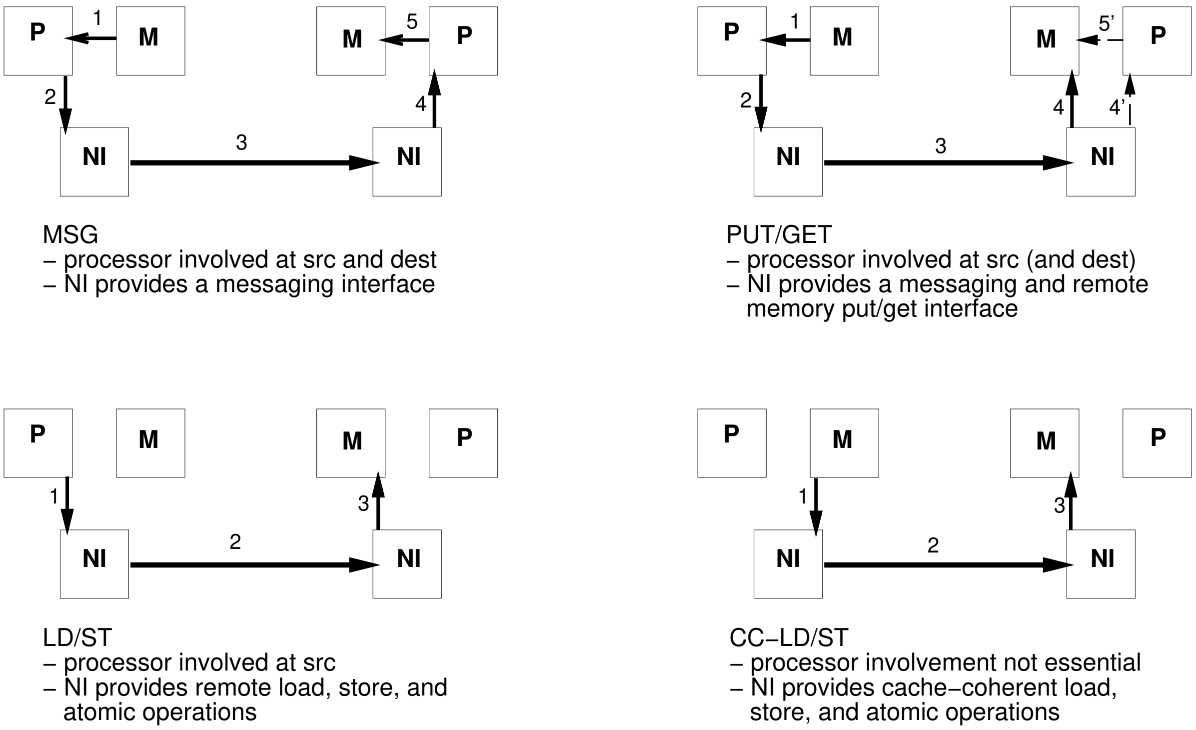

Figure 3.2 Four communication architecture interfaces.

[1] In this research, we restrict our attention to uniprocessor compute nodes.

- MSG is a two-sided communication architecture which supports a traditional messaging interface, requiring compute processor involvement both at the source and the destination in order to accomplish data transfer. The performance implication of compute processor involvement is that the processor needs to be responsive to incoming communication requests.

- PUT/GET is a one-sided communication architecture that allows a compute processor to directly write to (put) and read from (get) remote memory. Such support is present on the Princeton SHRIMP [123], the DEC Memory Channel [124], the HP Hamlyn [125], and versions of the IBM SP-x [126]. On such organizations, data transfer alone can be achieved without remote processor involvement, which is still required for atomic operations (shown by dashed lines).

- LD/ST is a one-sided communication architecture that differs from PUT/GET in supporting non-cache-coherent stores, loads and atomic operations against remote memory. Such support is available on the Cray T3D [40] and T3E [42]. On such organizations, both data transfer and atomic operations can be accomplished without involving the remote processor whose responsiveness is less of an issue.

- CC-LD/ST is a zero-sided communication architecture that differs from LD/ST in two ways. First, the load, store and atomic operations are cache-coherent, which results in a reduction in the effective cost of accessing remote memory. Second, data transfer does not require active participation from either processor: the network interface processor can directly access memory. Depending on the specific architectural parameters, this interface can model an I/O-mapped network interface processor as in the Myrinet network [127], or a hardware cache-coherent DSM multiprocessor with a custom protocol processor [57, 128–130].

The above interfaces represent discrete design points on the continuum between distributed-memory machines and a completely hardware-assisted cache-coherent shared-memory machine. We consider emulations of the above interfaces on our target platforms: the MSG, PUT/GET, and LD/ST interfaces can be emulated on both the Cray T3D and the SGI Cray Origin 2000, while the CC-LD/ST can only be emulated on the latter. The two emulations of each interface provide the same qualitative capabilities but with different quantitative characteristics. For example, since all data transfers in the SGI Cray Origin 2000 are cache-coherent, it represents a less aggressive emulation of the PUT/GET interface, and a more aggressive emulation of the LD/ST interface as compared to the corresponding emulations on the Cray T3D. Our focus in this research is more on evaluating the qualitative impact of architectural capabilities, and less on the quantitative differences.

3.5 Summary

In this chapter we have described problems contributing to the current poor performance of dynamic multithreaded computations on scalable parallel platforms, and our approach for addressing them.

The main performance challenges arise from the dynamic computation model and characteristics of irregular applications which require low overhead mechanisms for local efficiency, data locality, and load balance that are robust to computation irregularity. Previous approaches are inadequate, consisting of mechanisms which either incur high overhead or do not yield good performance in the presence of irregularity.

Our approach focuses on the design of low-overhead robust mechanisms for local and parallel efficiency. Local efficiency mechanisms yield performance conditioned upon run-time locality and thread placement, whereas parallel efficiency mechanisms for data locality and load balance yield performance robust to computation irregularity by exploiting knowledge of application semantics and properties of underlying target architectures. The local efficiency mechanisms are described in Chapter 4, and the parallel efficiency mechanisms are described in Chapters 5, 6, and 7.

CHAPTER 4

LOCAL EFFICIENCY

In this chapter we describe a set of thread management and communication mechanisms which ensure local efficiency conditioned upon run-time locality and load balance. These mechanisms adapt to the dynamic nature of computation and serve as a robust building block for parallel efficiency.

The chapter is organized as follows. Section 4.1 motivates the need for robust mechanisms by identifying requirements of an ideal local efficiency building block and demonstrating inadequacies in existing run-time techniques. Sections 4.2–4.4 describe the design and evaluation of two such techniques in the context of one of our target platforms, the Cray T3D. Section 4.2 describes an efficient thread management technique, *hybrid stack-heap execution*, which takes advantage of run-time locality situations to reduce overheads by coalescing computation into coarse-grained threads on the fly, while Section 4.3 describes a robust communication mechanism which provides high performance decoupled from local processor activity. The performance impact of these two mechanisms is evaluated in Section 4.4. Section 4.5 evaluates the impact of the mechanisms on our second target platform, the SGI Cray Origin 2000. Finally, Section 4.6 summarizes the key points of this chapter.

4.1 Overview

In this section, we motivate the need for robust local efficiency mechanisms. We first identify the requirements that must be satisfied by these mechanisms to serve as an appropriate building block for the parallel mechanisms. We then demonstrate using a compute-communicate microkernel that existing thread management and communication techniques described in Section 2.5.3 do not satisfy these requirements.

4.1.1 An ideal local efficiency building block

Our target execution platforms are based upon commodity-processing nodes which are connected to a scalable, low-latency, high-bandwidth communication network using a variety of communication architectures. These platforms do not contain hardware support either for the fine-grained interleaving of threads or for fine-grained communication, requiring both thread management and communication mechanisms to be implemented using software components. Thus, in order to serve as an effective building block for parallel efficiency, the local efficiency mechanisms must be designed to

1. minimize thread management and communication overheads,
2. exploit run-time locality and thread placement situations to improve performance, and
3. decouple activity on different processing nodes, minimizing the disruption on the local activity of a processing node by other processors.

The first requirement directly contributes to local efficiency, while the other two requirements allow the independent design of parallel efficiency mechanisms. The second requirement ensures that data locality and load balancing mechanisms can place threads and objects as appropriate at run time without having to explicitly aggregate or schedule threads for efficiency. The parallel mechanisms work with the understanding that the underlying local efficiency mechanisms can realize performance from good placement. The third requirement provides the data locality and load balancing mechanisms the freedom to initiate communication operations without having to explicitly schedule them to not disrupt ongoing computation.

4.1.2 Shortcomings of existing mechanisms

Although the base run-time mechanisms for thread management and communication described in Section 2.5.3 incur low overheads (taking advantage of heap-allocated user-level threads and a lean messaging interface respectively), they do not satisfy the above requirements preventing their use as a building block for parallel efficiency. We quantitatively demonstrate this shortcoming by considering the performance of a communicate-compute microkernel that uses the base mechanisms. The same microkernel is used later in the chapter to highlight the key performance advantages of robust local efficiency mechanisms.

4.1.2.1 Communicate-compute microkernel

The code for the microkernel is shown in Figure 4.1. The `main` routine creates a set of independent threads, `WorkThread`, on each processor. The `conc` annotation specifies that all the threads can be processed concurrently. The computation terminates when all threads on all processors complete. Each `WorkThread` thread consists of a data access phase where it gathers data values from (potentially remote) objects, followed by a phase where it computes using these values. Since data accesses are subject to the object-level concurrency control described in Section 2.1, each access creates a logical thread for processing the service request.

```
1   main() {
2      conc for (i=0; i<nprocs; i++)
3         conc for (j=0; j<nthreads; j++)
4            WorkThread();
5   }
6
7   WorkThread() {
8      // data access: obj is potentially remote
9      conc { x = valx(obj); y = valy(obj); z = valz(obj); }
10     // compute with data
11     compute(x, y, z);
12  }
```

Figure 4.1 Code structure of the communicate-compute microkernel (shown using ICC++ syntax). The execution behavior of the microkernel is characterized by the fraction and distribution of remote accesses in line 9 and the compute granularity and variance in line 11 (shown boxed).

This microkernel exhibits the general communication and computation characteristics of several irregular applications, such as molecular dynamics, hierarchical radiosity calculation, etc, where a thread first gathers data from a set of objects and then operates on the data. While the specific behavior can be controlled to model a variety of characteristics of dynamic multithreaded computations, we restrict our attention here to modeling the impact of fine-granularity threads and communication requests. In addition to quantifying the performance advantages of low-overhead primitives, the microkernel also tests whether primitives yield good performance in the face of interleaved communication and computation inherent in the multithreading model. The behavior of the microkernel is controlled by two parameters:

- Percentage of local accesses (line 9): This determines the amount of communication required by the microkernel. Many irregular applications have statically unpredictable communication behavior; consequently, the computation must adapt to run-time locality for efficiency.

- Compute granularity (line 11): This parameter affects the ratio of compute grain size to communication overhead. Natural expression of irregular applications often produces small granularity threads.

4.1.2.2 Performance of base run-time system mechanisms

To motivate the need for robust local efficiency mechanisms, we first examine the microkernel performance corresponding to the base mechanisms described in Section 2.5.3 (hereafter referred to as the Base run-time system). Given the lack of static information about thread blocking, the microkernel execution results in a physical thread for each `WorkThread` thread and for each data access request in line 9. Each such thread involves a `make_future`, any required communication (if the thread needs to be executed remotely),[1] creation, scheduling, and deallocation of the callee context, and touching the future along with suspension and restart (if required) of the caller. The compiler can still perform one optimization: the synchronization for all thread requests in lines 4 and 9 can be grouped, ensuring that the caller context suspends and restarts only once.

Figure 4.2 shows the performance of the microkernel using the Base run-time system on the T3D (left) and the Origin (right). Each plot shows the microkernel efficiency (percentage of run time spent on user computation) as the percentage of local accesses is varied from 0 (all remote) to 100 (all local). The curves are parameterized for different values of the compute granularity, τ, ranging from 10.0μs up to 5000.0μs. These parameter values reflect the execution behavior of several ICC++ applications described in Chapter 7. Each experiment was run on 16 processors of the machine, with the Origin run-time system configured for one processor per address space. The microkernel was instantiated with 2000 `WorkThread` threads per processor.

The general trend in both plots is as expected: execution efficiency increases with increases in the percentage of local accesses, and with increases in compute granularity. However, the following two observations indicate that mechanisms in the Base run-time system do not provide a good building block for parallel efficiency:

[1] We assume that the thread is executed local to the data object that it accesses.

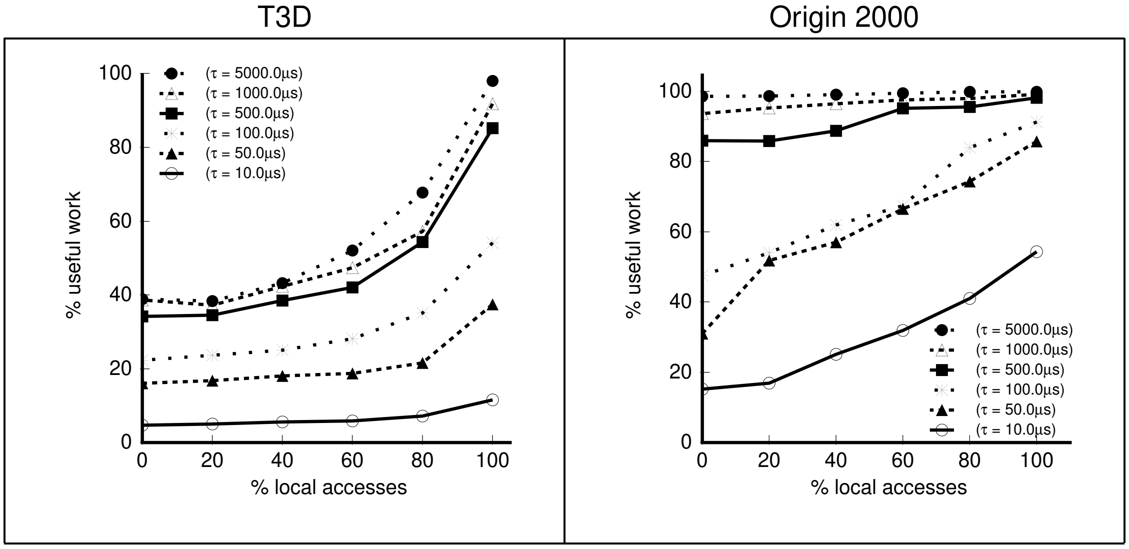

Figure 4.2 Microkernel efficiency on the T3D and the Origin for the Base run-time system mechanisms.

1. Even when a large percentage of accesses in line 9 of Figure 4.1 are to local objects, microkernel efficiency for small compute granularities is poor. For example, with 100% local accesses, the execution efficiency for a compute granularity of 10.0μs is only 11.5% on the T3D, and 54.3% on the Origin. The low efficiency primarily arises because lack of static information results in physical thread creation to complete data accesses, even when the target object happens to be local. Thus, the thread management mechanisms are unable to exploit run-time locality to improve performance.

2. The T3D performance plot shows that even with relatively large compute granularities, execution efficiency remains poor when a large fraction of data accesses are to remote objects. For example, even with compute granularities of 5000.0μs, execution efficiency remains below 50% if more than 40% of accesses are to remote objects. A closer examination shows that each processor spends a significant portion of its time waiting for communication operations to complete. These operations in turn are delayed because the destination processor is not responsive (with buffer management). Thus, the communication mechanisms do not decouple activity on different processing nodes, leading to poor performance if a processor is busy with computation when others wish to communicate with it.

The consequence of the low efficiency due to the above shortcomings is to limit the range of application characteristics that can be efficiently supported. In the next two sections, we describe two mechanisms which address the above shortcomings.

4.2 Robust Thread Management: Hybrid Stack-Heap Execution

In this section we describe an efficient thread management mechanism, hybrid stack-heap execution, which exploits run-time locality and thread placement situations to improve performance. As we saw in Section 4.1, although the base run-time primitives described in Section 2.5.3 enable efficient execution of finer granularity computations (particularly in comparison with portable run-time systems with more expensive primitives), they still incur sizable overhead for dynamic computations. These computations are less amenable to static analyses, so the compiler is typically unable to coalesce logical threads into sufficiently coarse-grained physical threads. The hybrid stack-heap execution mechanism addresses this shortcoming.

This section is organized as follows. Section 4.2.1 describes the design and implementation of the hybrid stack-heap execution mechanism, Section 4.2.2 evaluates the primitive costs of the mechanism, and Section 4.2.3 relates our approach with previous work.

4.2.1 Design and implementation

The hybrid stack-heap mechanism constructs coarser-grained physical threads "on the fly" from fine-grained logical threads, achieving high sequential efficiency when the thread accesses only local data objects and incurring minimal thread management overhead when the thread interacts with remote data objects or threads. Our technique provides a flexible interface for the run-time primitives to the compiler, enabling it to generate code which optimistically executes a logical thread sequentially on its caller's stack, *lazily* creating a different thread only when the callee computation needs to suspend or be scheduled separately. To separately optimize for the two modes of operation—sequential and parallel—the compiler generates two code versions for each thread body. One version executes off a stack-allocated activation frame and is optimized for sequential efficiency, accruing the advantages of procedure-call like efficiency for thread interactions when only local data is accessed. The other version operates from a heap-allocated context and is optimized to minimize thread scheduling and synchronization overheads. However, since a thread may not complete on the stack because of various blocking situations, the stack code version detects these situations at run time, lazily creates a heap context (corresponding to a physical thread) and resumes execution from the corresponding point in the heap code version. We define a hierarchy of interaction schemas for the stack versions of the thread code, which the compiler can optimize for efficiency. Table 4.1 summarizes these interaction schemas which are described below in detail.

Table 4.1 Various thread interaction schemas in the hybrid stack-heap execution model.

Version		Basic Operation
Heap		Most general schema, thread arguments/linkage through heap-allocated contexts
Stack	Nonblocking	Regular C call/return
	May-block	Regular call; detect and lazily create heap context on block
	Continuation-passing	Extension of may-block which allows forwarding on the stack

4.2.1.1 Heap version: Optimized for parallel execution

This version operates against a heap-allocated activation frame (context) and is a conservative implementation supporting thread suspension. Suspension may occur while interacting with remote data objects and threads, or while waiting for results from another blocked thread. Thread creation, scheduling, and synchronization are identical to that described in Section 2.5.3. The compiler optimizes the code to minimize the thread management overhead: several requests are issued in parallel and the touching of the corresponding futures is grouped so as to minimize the number of required restarts. In addition, the compiler carefully manages the amount of state that needs to be saved and restored across suspension points.

4.2.1.2 Stack versions: Optimized for sequential execution

Stack versions optimistically execute the thread code using a stack-allocated activation frame, lazily "falling back" to a heap context only when the thread needs to suspend. Thus, logical thread creation is essentially similar to a procedure call, incurring negligible overhead when the thread completes without blocking. However, when the thread blocks, we need to create the callee's heap context and set up the linkage between the caller and callee threads to allow thread execution to continue using the heap version. Depending on the thread interaction scenario, these operations require different information to be passed across the initial procedure call boundary and incur different fallback costs. To allow the compiler to optimize these costs based on available information, we define a hierarchy of stack schemas: nonblocking, may-block, and continuation-passing.

The *nonblocking* version is used when the compiler can prove that the called thread and all of its descendent calls will complete without blocking. When the compiler cannot prove this but knows that the callee thread does not require the caller's continuation (except to return a value), the *may-block* version is used. If the thread blocks, a heap context is lazily allocated to continue execution once it resumes. Finally, the *continuation-passing* version is used if the callee thread may require the continuation of a future in the (as yet uncreated) caller thread's context. In this case, we create both the callee's context as well as the continuation lazily. Our compiler selects the appropriate schema for each thread interaction based on a global flow analysis [18, 65] which conservatively determines the blocking and continuation requirements of each thread body [131]. A novel aspect of our hybrid stack-heap execution model is that it is implemented entirely in C, and consequently, is portable across a variety of parallel platforms. The discussion below describes how detection of successful completion and setting up of thread linkages in the event of fallback are handled by building on top of the traditional C procedure call/return linkage.

Nonblocking: Straight C call. When the compiler can ascertain that a thread will not block, all interactions are handled using a standard C call. This enables entire nonblocking subgraphs of the thread call tree to execute with no thread management overhead. The code sequences for the caller and callee threads are shown in Figure 4.3.

Caller	Callee

`...`
`rval = thread_code(...);`
`...`
	`ret_type thread_code(...) {`
	`   ...`
	`}`

Figure 4.3 Caller (left) and Callee (right) code sequences for the Non-blocking schema.

May-block: Lazy context (thread) creation. This stack version handles the case when a logical thread call may block. The calling schema, shown in Figure 4.4, distinguishes between the two outcomes: successful completion and a blocked callee thread. If the callee runs to completion, a NULL value is returned, and the caller thread extracts the actual return value from `rval`, a pointer to which is passed in as an argument to the call. If the callee thread blocks, the callee context (which itself was just created) is returned (in response to the C procedure call), enabling the caller to set up a linkage to the callee context. This is necessary because the linkage between caller and callee was implicit in the stack structure, and the caller must insert a continuation for the callee's return value into the callee context to preserve that linkage. Subsequently, the caller will, if necessary, create its own context, reverting to the parallel code version, save its state for restart from the heap version, and return its own context to its caller. Figure 4.5 shows the unwinding of stack frames when logical callee threads cannot complete their execution on the stack. In this case, the fallback code creates the callee's context, saves local state into it, and propagates the fall back by returning this context to its caller, which then sets up the linkage.

Thus, the may-block calling schema allows a sequence of may-block threads to run to completion on the stack, creating physical threads only as required by the run-time situations.

```
...
callee_context = thread_code(&rval,...);
if (callee_context != NULL) {
   callee_context->contin =
      make_continuation(rval);
   // propagate blocking
}
...
```

```
Context* thread_code(rval_ptr,...)  {
   ...
   if (non_blocking_arm) {
      *rval_ptr = val;
      return NULL;
   } else {        // fallback
      own_context = create_context();
      // save state to heap
      return own_context;
   }
   ...
}
```

Figure 4.4 Caller (left) and callee (right) code sequences for the may-block schema.

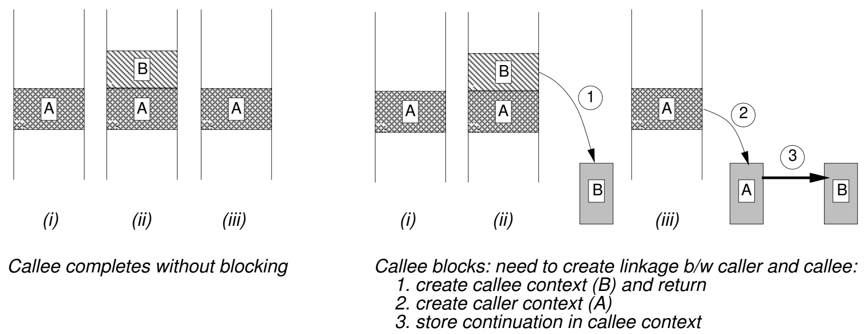

Figure 4.5 May-block interaction schema. The left figure shows successful completion, whereas the right figure shows the stack unwinding when the call cannot be completed.

Continuation-passing: Lazy continuation (and context) creation. Explicit continuation passing can improve the composability of concurrent programs [15, 16]. However, when continuation passing occurs, invocations on the stack are complicated because the callee may want its continuation. If the call is being executed on the stack, the callee's continuation is implicit. Since one of our goals is to execute forwarded invocations [132] on the stack, lazy allocation of the continuation is essential. As we shall see, allocation of a continuation also implies creation of the context in which the returned value will be stored.

The calling schema for the continuation passing case (see Figure 4.6) uses an additional parameter, `cinfo`, which, along with the `rval_ptr`, encodes information necessary to determine what to do should the continuation be needed. The `cinfo` information is simply passed along to support local forwarding, but if a thread tries to store the continuation or forward it off-node, it must be created. The `cinfo` parameter contains both information indicating whether the context containing the continuation's future has already been created and information sufficient to create both the context and the callee's continuation in case it has not.[2]

```
...
caller_context =
   thread_code(&rval,cinfo,...);
if (caller_context != NULL) {
   // save state to heap
   // propagate blocking
}
...
```

```
Context* thread_code(rval_ptr,cinfo,...)  {
   ...
   if ( need_continuation ) {      fallback
      caller_context =
         create_context(rval_ptr,cinfo);
      own_context = create_context();
      own_context->contin =
         make_continuation(caller_context,cinfo);
      // save state to heap
      return caller_context;
   } else {
      *rval_ptr = val;
      return NULL;
   }
   ...
}
```

Figure 4.6 Caller (left) and callee (right) code sequences for the continuation-passing schema.

When the continuation is not created, the thread that resolves the future simply stores the result through `rval_ptr`, and passes NULL return values back to its caller. The caller of the first continuation-passing method receives this NULL value and looks in `rval` for the result, thus executing the forwarded continuation completely on the stack. On the other hand, if the continuation is required by the callee,

[2] This typically requires information about the size of the caller context, and the return location within it.

cinfo is consulted. The fallback code first obtains the caller context (creating it, if necessary), and then the continuation for a new future at a location in the caller's context corresponding to rval_ptr and the return value offset stored in cinfo. The callee may now do whatever is desired with the continuation, finally passing the continuation's future's context back to its caller. The callee code in Figure 4.6 shows a situation where the callee also suspends, causing it to create its own context, save any state necessary, and insert the created continuation. Figure 4.7 shows the stack unwinding and linkages which need to be set up as part of the fallback. The reader is referred to [131] for additional details about the implementation.

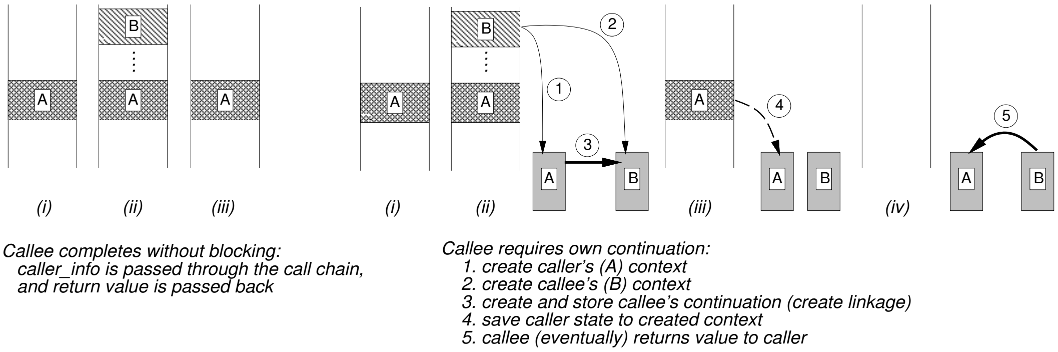

Figure 4.7 Continuation-passing interaction schema. The left figure shows callee completion without blocking, whereas the right figure shows the fallback when the callee requires the caller's continuation.

4.2.2 Primitive costs

Table 4.2 presents the cost of the hybrid stack-heap invocation mechanisms for various caller-callee scenarios on the Cray T3D. The costs are presented as overheads required in addition to the cost of a basic C function.[3] There are two components to this overhead: the first (shown in the left tables) corresponds to the situation when the callee thread completes on the stack, and the second (shown in the right tables) indicates the additional fallback cost when the callee stack frame must be unwound into the heap. Stack calls which complete without blocking have cost comparable to a basic C function call, and up to two orders of magnitude less overhead than the parallel (heap-based) thread interaction (320 cycles). The 6–8 additional cycles for sequential calls are due to call arguments, and passing the return value through memory, rather than in a register.

[3] On an Alpha (the node processor of the T3D), a C function call costs 25-40 cycles.

Table 4.2 Call and fallback costs (at the caller) for different caller-callee scenarios on the Cray T3D, in terms of overhead required in addition to a C function call. NB, MB and CP stand for nonblocking, may-block and continuation-passing sequential calling schemas, respectively.

CALL OVERHEAD (in Alpha cycles)

Calling schema for Caller		Calling schema for Callee: Parallel	Sequential: NB	MB	CP
Parallel		320	0	6	8
Seq.	NB	–	0	–	–
	MB	–	0	6	8
	CP	–	0	6	8

FALLBACK OVERHEAD (in μs)

Calling schema for Caller		Calling schema for Callee: Parallel	Sequential: NB	MB	CP
Parallel		–	0	0.15	0
Seq.	NB	–	0	–	–
	MB	–	0	2.00	0.30
	CP	–	0	3.50	1.80

The fallback overheads vary from 0.15 to 3.50 μs, depending on the specific caller-callee scenario. The unwinding costs are different for different caller and callee combinations because the different schemas place the responsibility for context creation and state saving at different places. These fallback overheads make explicit the tradeoff in using the sequential and parallel versions. The maximum fallback cost for any caller-callee pair[4] is comparable to the basic heap-based invocation (320 cycles), so optimistic execution using a sequential invocation first is effective in almost all cases. The same numbers also show that a sequential thread version can incur substantial overhead if it blocks repeatedly incurring multiple fallbacks; thus, reverting to the parallel method after the first fallback is a good strategy, especially if several synchronizations are likely.

4.2.3 Related work

Although there have been several hardware-based approaches for reducing thread management costs (e.g., providing multiple hardware contexts [133, 134] and integrating thread creation with message reception [135, 136]), we restrict our attention here to the more relevant software approaches which work with commodity hardware.

The idea of lazily creating threads as required by run-time situations can also be found in the work on Lazy Task Creation [24] and Leapfrogging [25] in the context of shared-memory machines, and Olden [9], Stacklets [26], and StackThreads [27] in the context of distributed-memory machines. The former two schemes were developed in the context of parallel languages with explicit futures [31], and

[4] The CP-MB fallback corresponds to context creations and scheduling for both caller and callee threads.

allow stealing previously deferred stack frames to adaptively control execution granularity and work distribution. Hybrid stack-heap execution differs in that it allows eager work distribution. The latter three schemes all allocate new threads whenever the current one blocks due to a remote operation. Thus, unlike hybrid stack-heap execution, they use the same mechanism to support both work distribution and latency-hiding and cannot optimize their code for either situation. A more recent version of StackThreads [137] borrows heavily from the work reported here, incorporating several of the same ideas in a standalone C library that does not require specialized compiler support.

4.2.4 Summary

The hybrid stack-heap run-time execution mechanisms permit the compiler to generate code which dynamically coalesces logical threads into larger-grained physical threads based on run-time locality and parallelism situations. By optimistically executing logical threads on the caller's stack using a specialized sequential code version, and lazily creating a physical thread only when the callee code needs to block, we minimize thread management overheads and accrue advantages of good sequential performance as appropriate.

The addition of the hybrid stack-heap execution mechanisms to the Base run-time system described in Section 4.1.2 produces a new version Base+Stack. This run-time system version exposes a flexible interface enabling the generated code to take advantage of run-time locality situations. Specifically, for the microkernel program, the compiler optimistically inlines the data accesses in line 9 predicated on locality and concurrency control checks. Remote accesses are handled efficiently by avoiding thread creation and scheduling overheads when the access can be completed without blocking for concurrency control reasons. The `WorkThread` code itself is invoked via a may-block stack schema, which implies that threads which end up accessing only local objects at run time execute completely on the stack, incurring only the cost of a C function call. Those threads which access truly remote objects lazily create heap contexts to complete their execution. The performance advantages of the hybrid stack-heap execution mechanism are evaluated in Section 4.4.

4.3 Robust Communication: Pull Messaging

In this section we describe an efficient messaging mechanism, pull messaging, which ensures processor decoupling by delivering performance robust over a range of dynamic communication characteristics. As we saw in Section 4.1, although the base run-time primitives described in Section 2.5.3 incur low overheads, they do not yield good performance when communication activities are unsynchronized with ongoing computation. The pull messaging mechanism addresses this shortcoming.

This section is organized as follows. Section 4.3.1 describes the design and implementation of the pull messaging mechanism in the context of the Cray T3D. Section 4.3.2 evaluates the primitive costs of the mechanism and demonstrates its robustness to output contention producing patterns. Finally, Section 4.3.3 compares our approach with previous work.

4.3.1 Design and implementation

The rationale for our approach stems from a closer look at Figure 4.2, which shows that the performance degradation with increased percentage of remote accesses is present only for the T3D implementation, not for the Origin one. The degradation arises from two qualitative differences between the distributed and shared memory implementations of messaging:

1. Buffer management in the distributed memory implementation couples source and destination activity: even though buffer allocation for individual sends can be accomplished without destination involvement, the destination eventually needs to participate to manage finite buffering resources (e.g., by reclaiming consumed buffers). On the other hand, this coupling is absent in the shared memory implementation.

2. Data transfer in the distributed memory implementation is eager or "push" based: the send completes when data is delivered at the destination. On the other hand, data transfer in the shared memory implementation is lazy, with the receiver "pulling" the message when ready.

The above differences explain the performance degradation seen by the T3D implementation. The source-destination coupling combined with eager data transfer implies that a slow destination (busy computing or unable to accept data fast enough because of output contention) can back up the network to other processors. This backup increases the send overheads for processors, which are now unable to inject messages into the network. The interleaving of communication and computation in the dynamic multithreaded model produces responsiveness delays at the destination, exacerbating the above.

Our solution, called pull messaging, integrates the key elements of a shared-memory messaging implementation—decoupling the source from destination activity, and "pull-based" lazy data transfer—in the context of a distributed memory machine. Specifically, it combines distributed message queuing (where the message data resides at the source processor) with lazy, receiver-initiated data transfer. Message send operations copy the message into a buffer at the source and then link in the message into a distributed queue associated with the destination. Message extract operations retrieve the message from the source processor's memory and invoke the corresponding handler. Pull messaging eliminates output contention and source-destination coupling since data are only moved when the receiver is ready to process it. An additional advantage is that buffer overflow happens rarely, only when a processor has sent many messages that have not yet been consumed. This type of fan-out problem is much easier to solve than fan-in. The essential component of the above technique is staging data transfer in accordance with destination activity (unlike traditional implementations which couple data transfer with source activity), and can be implemented on almost any distributed-memory parallel machine. However, realizing the performance advantages of pull-based messaging requires hardware support for remote memory access, which allows a source to link in the message without destination participation, and the destination to retrieve messages without source participation. Similar hardware support is present in several existing parallel machines (including the T3D, the Shrimp multicomputer [123, 138], and clusters based upon the DEC Memory Channel [124] and the HP Hamlyn [125,139] network interfaces), and a likely feature of future parallel machines.

Our implementation of pull-based messaging on the T3D utilizes hardware support for atomic-swap to link in messages into the distributed message queue, allowing send operations to proceed at the maximum swap rate. Message "pulling" by the destination is directly implemented using T3D's support for remote memory access. Thus, receive operations incur a round-trip latency; however, the T3D's prefetch queue can be used to mask this latency. Figure 4.8 shows the pull messaging implementation on the T3D. Each processor holds a tail pointer for its distributed message queue in a local memory location. Message transfer involves four steps: First, the source writes its message data into a local buffer and swaps a pointer to the buffer with the contents of the tail pointer on the destination (Step 1). This swap makes the local buffer the last entry in the destination's distributed message queue. The source then stores a pointer to the buffer into the memory address swapped out of the tail pointer (Step

2), linking the previous end of the queue to the local buffer. To receive, the destination uses prefetch operations to retrieve the message data (Step 3). On completion of the transfer, the destination processor invalidates the source buffer (Step 4), permitting its reuse. There are no races between steps 1 and 2 at the source and step 3 at the destination, because the destination can detect the end of the distributed message queue and simply wait (or go and do some other work) until more messages have been added.

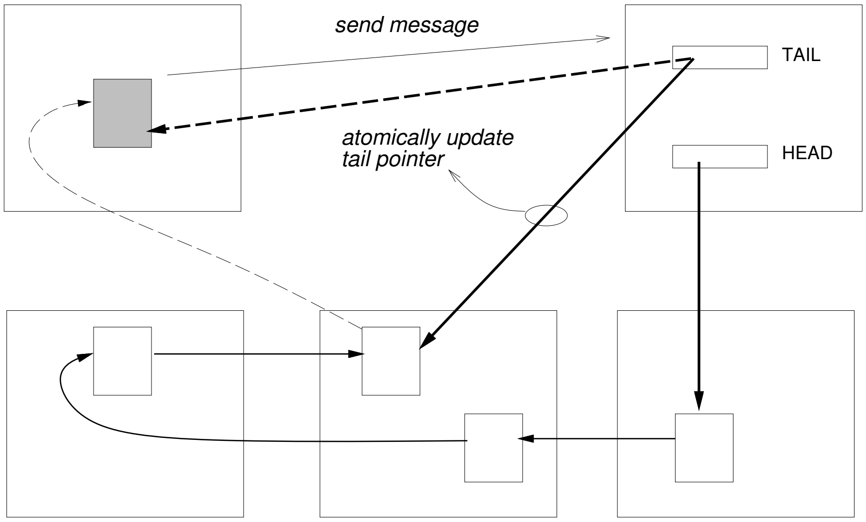

Figure 4.8 Distributed message queue implementation of pull-messaging. A message send links in the source-buffered message into a distributed queue (by atomically updating the tail pointer). A processor receives by “pulling” messages along its head pointer.

4.3.2 Primitive costs

Table 4.3 shows the source and destination overheads for the T3D push (fetch-and-increment based) and pull (atomic swap based) messaging implementations. The pull mechanisms are competitive for small messages, but pulling the message data results in increased overheads as compared with push messaging for messages larger than 64 bytes. This overhead difference is largely due to the high cost of interaction with the T3D prefetch queue: 0.1μs (15 cycles) to issue a single word fetch, and 0.15μs (20 cycles) to extract a word from the prefetch queue. Note however, that simple architectural improvements of the prefetch queue [84] can make the costs of pull messaging competitive with push messaging for all message sizes.

Table 4.3 Base source and destination overheads of the push and pull messaging schemes on the T3D. (all times in μs).

IMPLEMENTATION		MESSAGE SIZE (in bytes)						
		16	32	64	128	256	512	1024
Push Messaging (Fetch-and-increment based)	Src	1.84	1.85	2.03	2.84	3.53	8.26	14.65
	Dest	0.73	0.73	0.73	0.73	0.73	0.73	0.73
Pull Messaging (Atomic-swap based)	Src	1.49	1.56	1.67	1.93	2.49	4.13	6.31
	Dest	1.30	1.70	2.46	5.49	9.53	17.17	30.18

Pull messaging provides robustness to both temporal and spatial irregularity. We discuss its advantages over push messaging for spatial irregularity situations in the following section. Section 4.4 presents the quantitative impact of pull messaging for temporal irregularity situations.

4.3.2.1 Robustness to output contention

To examine the performance advantages of pull-based messaging versus push-based messaging, we measure the source overheads for various communication patterns (`all-to-all`, `random`, `hotspot`×`4`, `hotspot`×`16`, and `all-to-one`) which exhibit different degrees of spatial irregularity and output contention. The `all-to-all` pattern is a permutation pattern which generates moderate network contention but no output contention (fan-in), and is used as a baseline to evaluate the effect of output contention. In `random`, each processor picks a random destination and sends to it. Since several processors may pick the same destination, the pattern generates moderate fan-in. The `hotspot` patterns are the same as random, except that one node is designated a hotspot (processor 0) and receives a constant multiple of the traffic of the other nodes (we consider `hotspot`×`4` and `hotspot`×`16`). The hotspot intensity is proportional to the amount of fan-in. The `all-to-one` pattern represents the extreme hotspot situation where all the traffic is sent to a single processor. As we show below, pull messaging effectively insulates the source processor from activity in the network and at other processors, delivering performance that is virtually unaffected by the communication pattern.

Figure 4.9 shows the steady-state send and receive overheads of the push and pull implementations for four message sizes when the above traffic patterns are executed on a 64-node T3D. The plots are on a log-log scale and each point corresponds to the harmonic mean of source overheads over all 64 nodes. The point-to-point costs have been included for comparison. These graphs correspond to a specific compute delay between communication operations (13.2μs), but the performance trend remains essentially unchanged as compute granularity and its variance are modified.

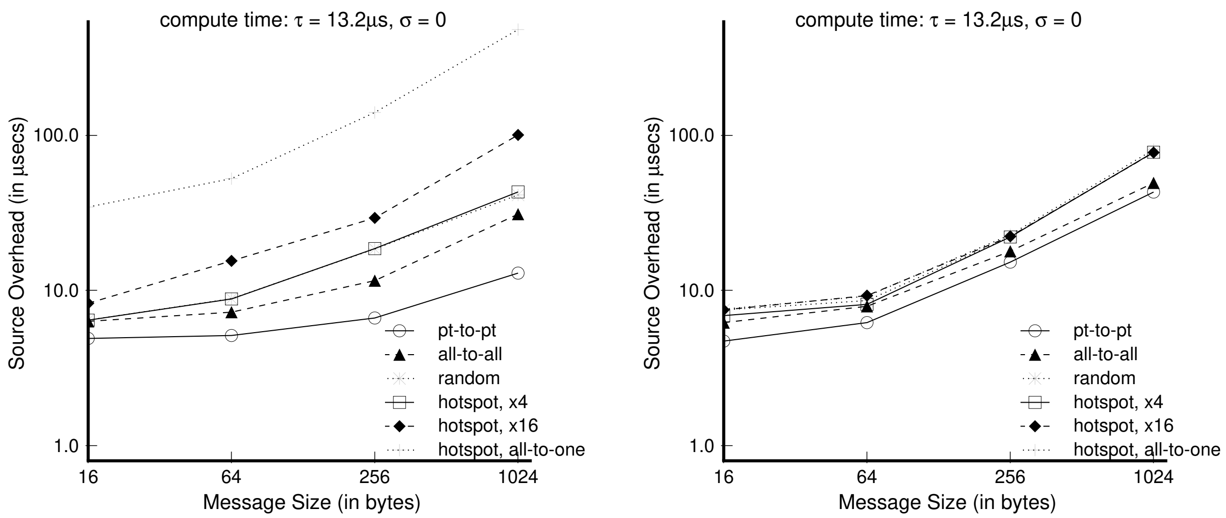

Figure 4.9 Source overheads for push (left) and pull (right) versions of messaging on the T3D for four message sizes.

The push messaging graph shows that the scheme is not robust over spatial irregularity. As the intensity of output contention increases (greater imbalance in message destinations or larger messages), source overheads increase by up to an order of magnitude as compared to the all-to-all case. For 1024 bytes, the source overheads for the point-to-point case are 13.5 μs, while they are 95 μs for the `hotspot`$\times$`16` pattern (3:1 degradation), and 455 μs for the `all-to-one` pattern (15:1 degradation). This performance loss is due to the classic hot spot contention [140], and will occur in any messaging layer which eagerly pushes messages to their destination.

In contrast, pull messaging delivers performance robust to spatial irregularity, as seen by the close clustering of the curves for different traffic patterns in Figure 4.9. As the hotspot intensity increases, there is no noticeable degradation in performance (the linear increase in send overheads with message size occurs because of the source copy). The major reason for pull messaging's robustness is the benefit of distributed queuing and lazy data movement. Distributed queuing can sustain large fan-in into hot nodes, supporting even all-to-one communication for extremely large machines with no degradation in performance. The hot node only "pulls" in message data as required, matching the data transmission rate to that feasible for the node. Thus, there is neither excessive data traffic nor enqueue traffic to interfere with other communication. In fact, we have found it effectively impossible to generate traffic loads which cause any input or output contention using pull-based messaging.

4.3.3 Related work

Although several researchers have focused on improving the performance of point-to-point communication, both from a hardware [134, 141] and a software [23, 142, 143] perspective, we restrict our attention here to performance optimizations for multiparty communication. Decoupling activities on different processors is only an issue in the latter.

Among hardware approaches, the Ultracomputer [144] provides a combining network which reduces output contention by combining fetch-and-op requests. NIFDY [145] is a proposed network interface which limits the number of outstanding messages between pairs of processors to improve network performance even for unbalanced traffic. Pull messaging achieves the same effect as these systems, but with general-purpose hardware: remote memory access and synchronization hardware is available in several current parallel machines.

Software approaches have focused for the most part on optimizing regular synchronous communication patterns, as evidenced by the work on collective operations [146], and all-pairs messaging [147]. In contrast, our pull-based messaging scheme provides robust performance in the presence of asynchronous and varied communication behaviors.

4.3.4 Summary

Pull messaging mechanisms enable the generated code to deliver communication performance that is robust over spatial irregularity (irregular, unbalanced traffic), and temporal irregularity (unsynchronized processor communication and computation phases), essentially decoupling activities on different processors. The mechanism relies on distributed message queuing, a form of source buffering which eliminates the detrimental coupling between sender and receiver, and lazy receiver-initiated data transfer which eliminates output contention by matching the data transfer rate to that sustainable by the node.

The addition of pull messaging on the T3D to the run-time primitives described in Sections 4.1.2 and 4.2 produces the Base+Pull and Base+Stack+Pull run-time versions. Section 4.4 presents the performance impact of pull-messaging mechanisms, evaluated using the microkernel program.

4.4 Performance of Mechanisms on the Cray T3D

In this section, we quantify the performance impact of the two local efficiency mechanisms described in Sections 4.2 and 4.3. To measure the individual impact of the mechanisms, we compare the microkernel performance on the T3D corresponding to the Base run-time system (previously described in Section 4.1.2) with the Base+Stack and Base+Pull run-time systems, respectively. We also measure the performance corresponding to Base+Stack+Pull which represents the aggregate impact of both mechanisms.

As in Section 4.1.2, all experiments were run on 16-node machine configurations. Figure 4.10 shows the microkernel execution efficiency for the various run-time system versions on the T3D, as the percentage of local accesses is varied from 0% (all remote) to 100%(all local). The six graphs correspond to different values of the compute granularity. The following conclusions can be drawn from the results:

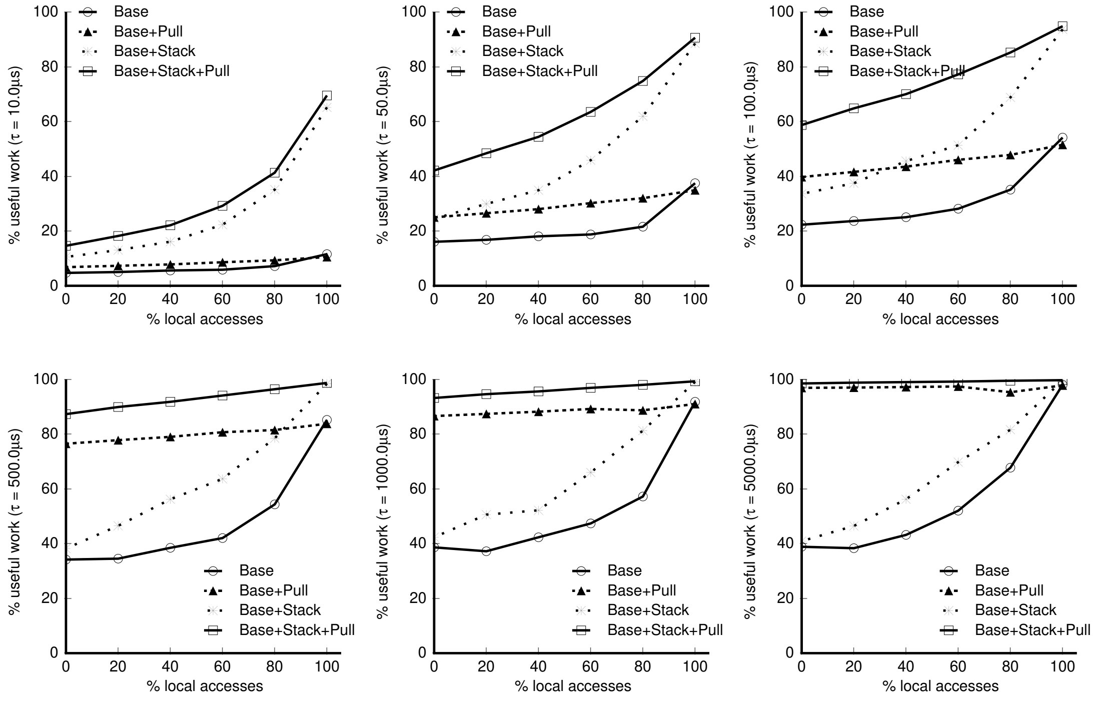

Figure 4.10 Impact of local efficiency mechanisms on the Cray T3D.

1. *Impact of hybrid stack-heap execution:* The performance increment between the Base+Stack and Base versions and between the Base+Stack+Pull and Base+Pull versions shows that hybrid stack-heap execution mechanisms can *increase execution efficiency by up to 50%*. For a specific compute granularity, the advantages from hybrid execution mechanisms increase with run-time locality, validating our earlier claim that the mechanism enables higher performance by adapting to run-time locality situations. The mechanism enables high performance for a broader range of compute granularities; for example, 70% efficiency can be achieved with granularities as low as 10μs. Since the run-time system can only decrease overheads, the Base scheme catches up for larger compute granularities.

2. *Impact of Pull Messaging:* The performance increment between the Base+Pull and Base versions and between the Base+Stack+Pull and Base+Stack versions, shows that pull-based messaging *can increase execution efficiency by up to 50%* by decreasing the coupling between the sender and receiver processors. This decoupling becomes less important as the amount of communication decreases, so the advantages from pull messaging decrease with increased run-time locality. For a fixed fraction of local accesses, the advantages from pull-messaging increase with increases in compute granularity. This behavior arises from the microkernel structure where increasing compute granularity also increases the likelihood that a destination processor will be unresponsive to service a communication request. Performance degrades for push messaging because of sender-receiver coupling, while pull messaging eliminates this coupling to deliver robust high performance.

In addition to the specific trends described above, one trend can be observed across all the graphs. The hybrid stack-heap execution model is effective in improving the efficiency of computations with small granularity and a high percentage of local accesses. The pull messaging mechanisms targets the other end of the parameter space, improving efficiency for large granularity computations which have a significant fraction of remote accesses. Thus, for the microkernel, the two mechanisms enable computations with significantly smaller compute granularity and larger amounts of communication to be efficiently executed.

One way of quantifying the above advantage is to identify regions of the parameter space for which a particular local efficiency mechanism helps achieve a specific minimum execution efficiency. For example, Figure 4.11 shows regions of the parameter space corresponding to compute granularity and percentage of local accesses, where the microkernel execution achieves at least 70% efficiency. Note that the compute granularity is plotted on a log scale. The $\geq 70\%$ efficiency region with the Base version is shown in black at the upper-right corner of the graphs; the addition of the two mechanisms increases this region with the increments corresponding to the appropriately labeled gray and hashed portions.

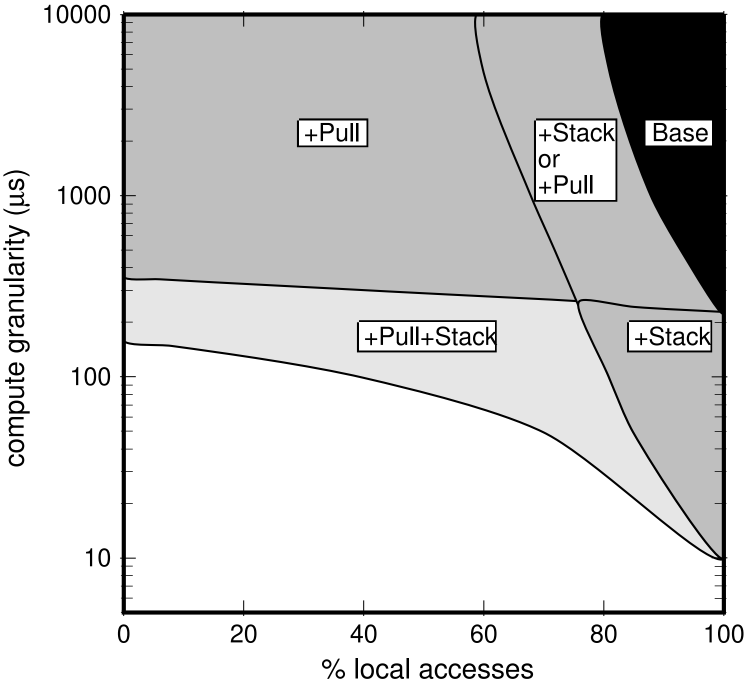

Figure 4.11 Regions of the compute granularity and fraction of local accesses parameter space where the microkernel achieves $\geq 70\%$ efficiency for different run-time system versions on the Cray T3D. The efficiency region achieved by only the Base run-time system is shown in black; adding the mechanisms increases this region (the increments correspond to the appropriately labeled hashed and gray portions).

The Base version requires compute granularities greater than 300 μs and greater than 80% local accesses.[5] The addition of the hybrid stack-heap mechanisms increases the acceptable efficiency region to include the hashed portions, marked +Stack and +Stack or +Pull. This enables 70% execution

[5] The 70% efficiency region requires a large fraction of local accesses even for relatively large compute granularities because the sender-receiver coupling worsens as compute granularity is increased, increasing the likelihood that a receiver is unresponsive.

efficiency with granularities as low as $10\,\mu$s and fewer local accesses (60%). The addition of the pull communication mechanisms increases the region further (shown by the hashed portion marked +Pull) allowing microkernel executions with even 0% local accesses to achieve $\geq 70\%$ efficiency. Finally, the two mechanisms together add the solid gray region marked +Pull+Stack, allowing 70% execution efficiency to be achieved with granularities as low as $10\,\mu$s (for 100% local accesses) and $150\,\mu$s (for 0% local accesses).

Summary. The two local efficiency mechanisms increase the flexibility of achieving high performance, enabling even computations with smaller compute granularities and a larger fraction of remote accesses to achieve high-efficiency levels. The plot also highlights the relative merits of the two mechanisms. Hybrid stack-heap execution enables small granularity threads to be executed efficiently, as long as the execution has sufficient run-time locality. On the other hand, pull-messaging mechanisms increase the locality region, which can be executed efficiently for a particular compute granularity. Thus, the mechanisms provides a high degree of flexibility to the parallel efficiency mechanisms which can concentrate on trying to obtain high levels of data locality and load balance.

4.5 Performance of Mechanisms on the SGI Origin 2000

In this section we describe performance of the local efficiency mechanisms on our other target platform, the SGI Origin 2000. Since the SGI Origin has several architectural differences from the Cray T3D, we first discuss the applicability of the two mechanisms, and then present performance benefits of applicable mechanisms. As we shall see, hybrid stack-heap execution is still applicable and yields comparable performance and robustness improvements. On the other hand, Origin's support for hardware cache-coherence reduces the need for pull messaging mechanisms and indicates the improved suitability of recent parallel architectures for decoupled communication.

4.5.1 Applicability

Robust thread management: Hybrid stack-heap execution. Since the hybrid stack-heap execution model is implemented entirely in C, the T3D implementation is portable to the Origin without any changes.

Table 4.4 presents the cost of the hybrid stack-heap invocation mechanisms for various caller-callee scenarios on the Origin 2000. As before, the costs are presented as overheads required in addition

Table 4.4 Call and fallback costs (at the caller) for different caller-callee scenarios on the SGI Origin 2000, in terms of overhead required in addition to a C function call. NB, MB and CP stand for nonblocking, may-block and continuation-passing sequential calling schemas respectively.

CALL OVERHEAD (in R10000 cycles)

Calling schema for Caller		Calling schema for Callee: Parallel	Sequential: NB	Sequential: MB	Sequential: CP
Parallel		104	0	6	8
Seq.	NB	–	0	–	–
	MB	–	0	6	8
	CP	–	0	6	8

FALLBACK OVERHEAD (in μs)

Calling schema for Caller		Calling schema for Callee: Parallel	Sequential: NB	Sequential: MB	Sequential: CP
Parallel		–	0	0.03	0
Seq.	NB	–	0	–	–
	MB	–	0	0.34	0.06
	CP	–	0	0.90	0.50

to the cost of a basic C function.[6] Similar to the T3D implementation, stack calls, which complete without blocking have cost comparable to a basic C function call, and up to two orders of magnitude less overhead than the parallel (heap-based) thread interaction (104 cycles). The fallback overheads vary from 0.03 to 0.90 μs depending on the caller-callee scenario. As before, the maximum fallback cost for any caller-callee pair is comparable to the basic heap-based invocation (104 cycles), so optimistic execution using a sequential invocation first is effective in almost all cases.

Robust communication: Pull messaging. The base implementation of messaging on a shared memory platform already exhibits the two major characteristics of pull messaging:

1. Buffer management is decoupled from destination activity: the availability of atomic operations and a shared address space ensure that message buffers can be allocated from a common pool without requiring destination processor participation.

2. Data transfer is pull based: since all communication in the system is built on top of the underlying cache-coherence operations, the message data is actually transferred to the destination only when the destination is ready to process the message.

Consequently, pull messaging mechanisms provide negligible additional benefits. These characteristics of the base messaging implementation on the Origin are an indication of the better decoupling properties of hardware shared memory parallel architectures as compared to distributed memory architectures.

[6] On an R10000 (the node processor of the Origin), a C function call costs 15-20 cycles.

4.5.2 Performance

Figure 4.12 shows the microkernel execution efficiency for the Base and Base+Stack run-time system versions on the Origin, as the percentage of local accesses is varied from 0% (all remote) to 100%(all local). As before, all experiments were run on 16 nodes of the machine. The six graphs in the figure correspond to different values of the compute granularity.

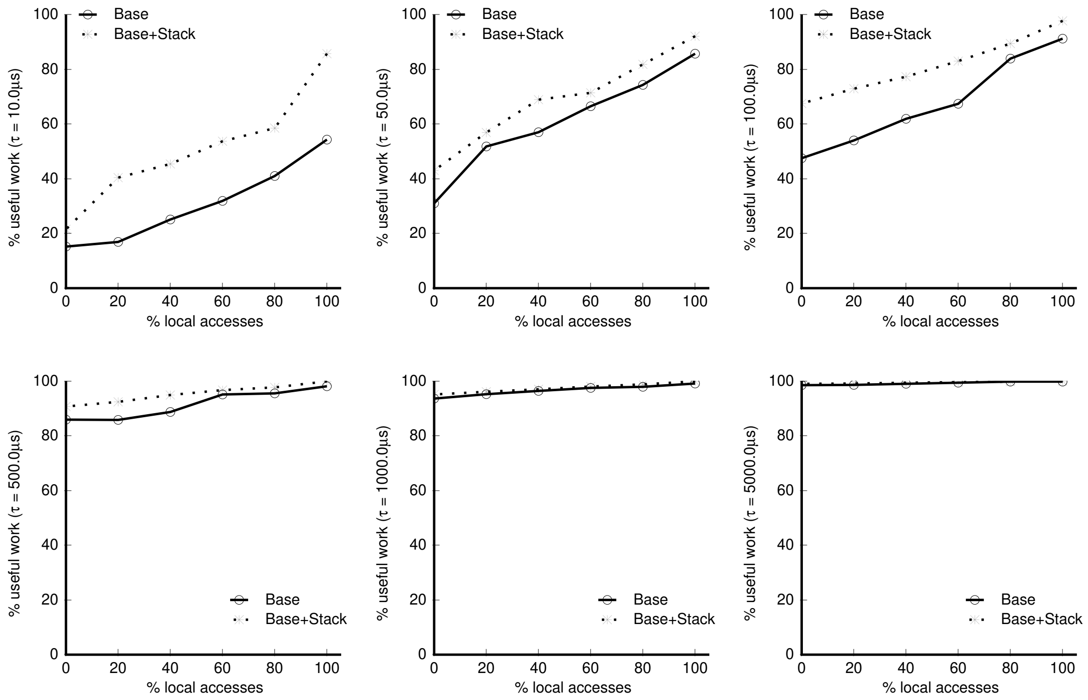

Figure 4.12 Impact of run-time system mechanisms on the SGI Origin 2000.

As on the T3D, the hybrid stack-heap execution mechanism improves microkernel execution efficiency by up to 50%, permitting 70% efficiency to be achieved with granularities as low as 10 μs. As earlier, for a specific compute granularity, the advantages from hybrid execution mechanisms increase with run-time locality, validating our earlier claim that the mechanism enables higher performance by adapting to run-time locality situations. For higher compute granularities, the Base scheme also achieves high efficiency. When contrasted with Figure 4.10, the graphs show the better decoupling abilities of the base messaging implementation on the Origin: execution efficiency increases in direct proportion to the compute granularity.

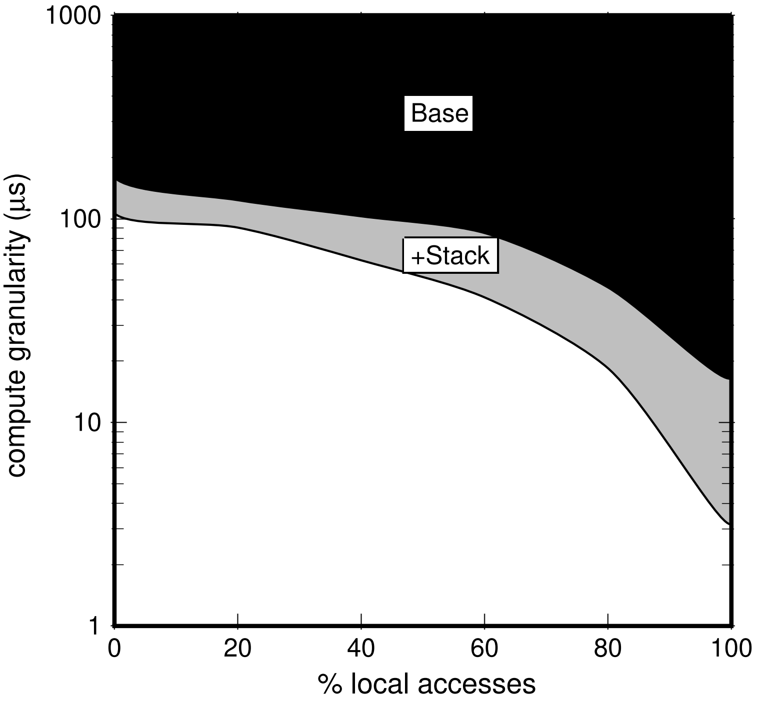

Figure 4.13 Regions of the compute granularity and fraction of local accesses parameter space where the microkernel achieves $\geq 70\%$ efficiency for different run-time system versions on the Origin. The efficiency region achieved by only the Base run-time system is shown in black; adding the stack-heap mechanism increases this region (the increment correspond to the hashed portion).

To quantify the additional set of applications that can be efficiently executed using the hybrid stack-heap execution mechanism, Figure 4.13 shows as an example, regions of the parameter space where the microkernel execution achieves $\geq 70\%$ efficiency. To achieve $\geq 70\%$ efficiency, the Base version requires compute granularities greater than 16.7 μs when all the accesses are local and greater than 162.5 μs when all the accesses are remote. Unlike the T3D case, the communication mechanisms in the Base run-time system are better suited to support dynamic multithreaded computations. The addition of hybrid stack-heap mechanisms increases the acceptable efficiency region to include the hashed portion, marked stack. This enables 70% execution efficiency with granularities as low as 3.2 μs (for 100% local accesses) and 108.3 μs (for 0% local accesses).

Summary. The hybrid stack-heap execution mechanism is equally effective on the Origin 2000, exploiting run-time locality to improve thread management performance. On the other hand, the pull messaging mechanism yields negligible additional benefits over the base shared memory messaging implementation, which already decouples activities on different processors.

4.6 Summary

In this chapter we have described two thread management and communication mechanisms which ensure local efficiency conditioned upon run-time locality and load balance. These mechanisms adapt to the dynamic nature of computation, and serve as a robust building block for the parallel efficiency mechanisms described in the next two chapters.

The hybrid stack-heap execution mechanism permits the compiler to generate code which dynamically coalesces logical threads into larger-grained physical threads based on run-time locality and parallelism situations. By optimistically executing logical threads on the caller's stack using a specialized sequential code version, and lazily creating a physical thread only when the callee code needs to block, thread management overheads are minimized and advantages of good sequential performance are accrued as appropriate.

The pull-messaging mechanism enables the generated code to deliver communication performance that is robust over spatial irregularity (irregular, unbalanced traffic), and temporal irregularity (unsynchronized processor communication and computation phases), essentially decoupling activities on different processors. The mechanism relies on distributed message queuing to decouple source and destination activities, and lazy receiver-initiated data transfer to eliminate output contention.

Individually, the mechanisms improve execution efficiency of a compute-communicate microkernel by up to 50%. Collectively, the impact of the mechanisms is to facilitate efficient execution for applications with smaller computation granularities and more communication operations. This added flexibility permits the parallel efficiency mechanisms described in Chapters 5 and 6 to be developed independently of local efficiency concerns.

CHAPTER 5

DATA LOCALITY

In this chapter, we describe a latency-tolerant caching framework, view caching, for improving data locality of dynamic multithreaded computations. The framework is motivated by the poor performance of existing object-shared memory implementations in such computations with fine-granularity threads and temporal and spatial irregularity. Our solution utilizes knowledge of application data access semantics to construct custom coherence protocols which require reduced synchronization and show increased tolerance to computation irregularity.

This chapter is organized as follows. In Section 5.1, we discuss the problem in more detail and provide an overview of our solution. In Section 5.2, we present the experimental context for this research, describing a state-of-the-art object-consistent shared memory system and a sharing microkernel used to evaluate different data locality techniques. Section 5.3 discusses implementation of the object shared memory system on different communication architectures. The microkernel performance of these implementations is evaluated in Section 5.4, and motivates the view caching framework described and evaluated in Sections 5.5 and 5.6, respectively. In Section 5.7, we discuss related work and Section 5.8 summarizes our contributions.

5.1 Overview

Data locality mechanisms reduce communication operations required to access potentially remote objects. Our focus here is on data locality for enhancing local efficiency; Chapter 6 discusses data locality for load balance. We restrict our attention to distributed object shared memory (DSM) techniques, where objects are dynamically cached on demand. Additionally, since not all the target platforms described in Section 2.2.2 provide hardware support for shared memory, we focus on software DSM techniques. Alternate data locality approaches which statically collocate objects "close" (in a memory hierarchy sense) to threads that access them are not feasible given limitations of static analyses in the context of dynamic multithreaded computations.

In this section, we first describe the performance challenges faced existing distributed object shared memory techniques, and then provide an overview of our approach.

5.1.1 The problem

Two characteristics of dynamic multithreaded computations (discussed in Section 3.1) limit the effectiveness of existing software DSM techniques:

- Threads exhibit wide variations in granularity
- Implementations of such computations exhibit temporal and spatial irregularity of the kind described in Section 3.1.3

When combined with the fact that software DSM techniques implement coherence actions via coroutining, which relies on explicit messages being transferred between nodes, these characteristics pose significant performance challenges. Wide granularity variations imply the presence of some fine-granularity threads, requiring low-overhead DSM implementations for efficiency. Temporal and spatial irregularity implies that coherence messages may face arbitrary processing delays because communication operations are not synchronized with computation, requiring DSM implementations that can tolerate such delays.

Unfortunately, existing DSM systems are poorly equipped to handle these challenges, particularly tolerance to responsiveness delays. Most DSM implementations are based upon synchronous (request-reply) operations, modeled after hardware cache-coherence protocols. While such protocols yield good performance with responsive message processing (as with dedicated cache controllers), they result in large idle times when that is not the case. Even state-of-the-art DSM implementations [77–79, 101], that rely on weak consistency models to reduce coherence traffic, retain the synchronous (request-reply) nature of hardware cache-coherence protocols, requiring responsive processing for performance.

Figure 5.1 demonstrates this behavior in the context of cache miss processing of a read request. The requesting processor (requester) sends the request to the processor which keeps track of object copies (directory). Satisfying the request may, in turn, require that an outstanding copy of the object be invalidated (on previous owner). Spatial and temporal irregularity implies that the request and

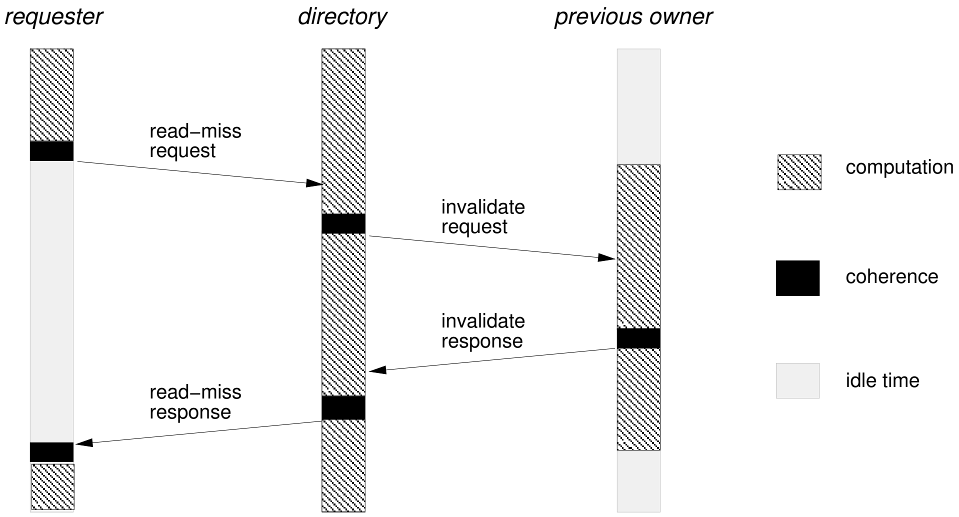

Figure 5.1 Traditional software DSM protocols incur increased idle time in the context of dynamic multithreaded computations.

invalidation messages are subject to processing delays (because the target processor is busy computing, or processing other requests).[1] The consequence is that poor responsiveness degrades performance by increasing the time a processor idles awaiting completion of a coherence operation.

Among other DSM approaches, techniques [148, 149] which optimize synchronous protocols by predicting object access patterns are less effective given a dynamic context. Recent systems [28, 30] which allow application customization of protocols provide mechanisms for improving performance, but offer little insight into how to build protocols for tolerating unresponsive processors. Our solution utilizes similar mechanisms, but differs in that it provides a framework of custom latency-tolerant coherence protocols which yield performance robust over unresponsiveness.

5.1.2 Overview of solution

The key problem in improving DSM performance for dynamic computations is to prevent unresponsive processors from affecting the performance of coherence protocols. Our solution exploits the availability of sophisticated communication architecture features, and knowledge of application data access semantics, to address this problem from both a hardware and a software perspective.

[1] One approach to avoid these delays is to have the coherence messages interrupt the destination processor. However, given the high cost of interrupts in current-day microprocessors, this alternative precludes fine-grained sharing.

To understand the extent to which richer features found in the communication architectures described in Section 2.2.2 affect performance, we first evaluate the performance of a state-of-the-art object-consistent DSM system on various communication architectures. Our results in Section 5.4 show that despite the availability of low-overhead and responsive operations, existing DSM systems are unable to achieve robust performance for dynamic computations.

The second component of our solution addresses this lack of robustness, which results primarily from the synchronous nature of coherence protocols in existing DSM systems. Exploiting application knowledge of data-access semantics, we develop a suite of latency-tolerant coherence protocols which require less synchronization among the participating processors, and hence are more tolerant to processing delays. In addition, our solution, called *view caching*, leverages responsive communication architecture features to implement these custom protocols efficiently, yielding robust sharing performance.

Sections 5.3 – 5.6 describe the above aspects of our solution in more detail, in the context of a specific object-consistent DSM system presented next.

5.2 Experimental Context

In this section, we present the experimental context for evaluation of the view caching framework. We first describe a state-of-the-art object shared memory system, the C Region Library, which provides the implementation context for this work. We then describe a synthetic sharing microkernel which provides the evaluation basis.

5.2.1 Object-consistent DSM: The C Region Library

The C Region Library (CRL) [79] is an efficient implementation of a state of the art object shared memory system. Although the source code of CRL Version 1.0 is available, we have chosen to reimplement the library. Our implementation, while retaining the same basic interface and the internal structure of the original, differs in two significant ways. First, our implementation is designed to fully exploit differing communication architectures. Second, it has been extended with view caching coherence protocols. A detailed description of the changes is deferred to Sections 5.3 and 5.5 respectively. Here, we present the basic structure of the CRL library, discussing in turn, the object consistency model, the CRL interface, and its implementation.

5.2.1.1 Object consistency

CRL maintains coherence at object granularity in accordance with the object consistency model. A DSM's consistency model is a contract between the application and a DSM implementation and defines the latest time when copies of an object need to be made consistent.

Object consistency is a weak consistency model which guarantees that the most recent updates to a shared object are visible only at the time of object access. This allows the underlying implementation to delay propagating updates performed by a thread until another thread accesses the same object. Examples of such systems in literature include Shared Regions [150], Midway [77], SAM [78], and CRL [79]. All these systems demarcate the region in which a shared data object is accessed with user annotations.

In the context of the concurrent object-oriented programming model described in Section 2.1, methods naturally define the extent of access to a shared object. For such a model, the object-consistent DSM system guarantees that all updates are visible at the start of method execution.

5.2.1.2 The CRL interface

This section describes the CRL interface, which is a set of calls used internally in the Concert run-time system to create and access shared objects. Table 5.1 summarizes this interface. The unit of coherence in CRL is a user-defined arbitrary length object. A sequential object is upgraded to a shared object (which can now be cached) using the `rgn_create` call, whose effect is to associate a caching metadata structure with the sequential object. The `rgn_destroy` call disassociates this metadata structure. The rest of the CRL interface decouples the two components of caching – renaming and coherence. The `rgn_map` and `rgn_unmap` calls are responsible for explicitly mapping and unmapping the shared object into the local address space, thereby providing renaming. Each access to the mapped object must be enclosed within `rgn_start_{read,write}` and `rgn_end_{read,write}` calls. These calls trigger coherence actions which ensure that write operations against the region are delayed until all reads have completed, and that no reads proceed until the ongoing (single) write has finished.

Table 5.1 Summary of the object-consistent CRL interface.

OPERATION	FUNCTION
Allocation	`rid_t rgn_create(int size, void *obj)` Associate metadata with `obj`
	`void rgn_delete(rid_t id)` Delete metadata
Renaming	`void* rgn_map(rid_t id, int otype)` Map a region of type `otype` associated with region identifier `id` into the local address space
	`void rgn_unmap(void *addr)` Unmap a mapped region
Access	`void rgn_start_read(void *addr)` Initiate a read operation on a region
	`void rgn_end_read(void *addr)` Terminate a read operation on a region
	`void rgn_start_write(void *addr)` Initiate a write operation on a region
	`void rgn_end_write(void *addr)` Terminate a write operation on a region

5.2.1.3 The CRL implementation

This section describes the implementation of renaming and coherence actions in CRL. Renaming is implemented using a hashing based lookup scheme to resolve mappings between global region identifiers and the corresponding local objects. Coherence is implemented using an invalidation-based directory protocol. The directory information for a shared region is maintained on a distinguished processor, the *home*. Access requests that cannot be satisfied locally are forwarded to the home node, which invalidates conflicting copies and updates the directory information prior to forwarding an updated copy of the region to the requesting node.

The CRL coherence protocol can be described by state machines at the home and remote nodes. The interested reader is referred to [151] for a detailed description of the state machines. Here, we focus only on providing sufficient context to understand the different implementations discussed in the rest of this chapter. The states and events in the state machines are shown in Tables 5.2 and 5.3, respectively. The protocol requires eight states for each object at the home and remote nodes to distinguish different access states. The region starts off by default in the HomeExclusive state on the home node, and RemoteInvalid states on the remote nodes. Unlike hardware cache coherence protocols, protocol states are also needed to keep track of ongoing read and write operations because, given the nature of the interface, access permission cannot be revoked until all ongoing operations have completed. These additional states include HomeExclusiveRip, HomeExclusiveWip, HomeSharedRip, RemoteSharedRip, RemoteModifiedRip, and RemoteModifiedWip.

State transitions are triggered by events (protocol messages). The `rgn_start_{read,write}` and `rgn_end_{read,write}` call events initiate state transitions on the local node. In addition, there are two kinds of protocol messages between the home node and the remote node. In the CRL protocol two remote nodes do not exchange messages directly, all multiple-node-protocol transactions are required to go through the home node.[2] Messages from the remote node to the home node include access requests (MsgSharedReq, MsgExclusiveReq, and MsgModifyReq), and acknowledgements to invalidation requests from the home node (MsgInvalidateAck{Data}, and MsgRelease). Messages from the home node to the remote node include invalidation request messages (MsgRInvalidate, MsgWInvalidate), and responses to the access requests (MsgSharedAckData, MsgExclusiveAckData, MsgModifyAck, and MsgModifyAckData).

The simple interface and coherence protocol together contribute to an efficient implementation [79]. Specifically, the protocol is implemented using indirect calls associated with local call events and message events. The event identifier is used to index through a function table which contains an entry for each possible kind of event. Each state of the state machine is represented by a different function table. To simplify the writing of event handlers, the underlying implementation ensures that each handler executes atomically.

5.2.2 Sharing microkernel

This section describes the structure of a sharing microkernel used to evaluate the shared object memory systems discussed in this chapter. A synthetic microkernel allows the easy modeling of a range of application behaviors encountered in irregular and adaptive applications [4, 5]. The performance issues highlighted by microkernel experiments will be corroborated later for whole application programs in Chapter 7.

[2] The CRL protocol does not exhibit the three-party optimization, common to several hardware coherence protocols [50,58] and some recent software protocols [108], where a modified copy is directly forwarded from the previous (nonhome) owner to the current requester. However, this optimization still retains the synchronous nature of the coherence protocol and is subject to a similar degradation due to unresponsive processors.

Table 5.2 Summary of base CRL protocol states.

Home-side Protocol States

State	Description
HomeExclusive	This node (the home node) has the only valid copy of the region
HomeExclusiveRip	Like HomeExclusive, plus one or more read operations are in progress
HomeExclusiveWip	Like HomeExclusive, plus a write operation is in progress
HomeShared	Both the home node and at least one other node have valid copies of the region (with shared access)
HomeSharedRip	Like HomeShared, plus one or more read operations are in progress
HomeInvalid	A remote node has a valid copy of the region (with exclusive access)
HomeIip	An invalidation of remote copies is in progress (to obtain exclusive access)
HomeIipSpecial	An invalidation of remote copies is in progress (to obtain shared access)

Remote-side Protocol States

State	Description
RemoteInvalid	This node does not have a valid copy of the region
RemoteInvalidReq	Like RemoteInvalid, but a request to acquire a valid copy of the region is pending
RemoteShared	This node has a valid copy of the region (with shared access)
RemoteSharedRip	Like RemoteShared, plus one or more read operations are in progress
RemoteSharedReq	Like RemoteShared, plus a request to upgrade to exclusive access is pending
RemoteModified	This node has a valid copy of the region (with exclusive access)
RemoteModifiedRip	Like RemoteModified, plus one or more read operations are in progress
RemoteModifiedWip	Like RemoteModified, plus a write operation is in progress

Table 5.3 Summary of base CRL protocol events.

Home-to-Remote Protocol Messages

MESSAGE	DESCRIPTION
MsgRInvalidate	Invalidate a remote copy of a region (to obtain a shared copy)
MsgWInvalidate	Invalidate a remote copy of a region (to obtain an exclusive copy)
MsgSharedAckData	Acknowledge a request for a shared copy of the region
MsgExclusiveAckData	Acknowledge a request for an exclusive copy of the region
MsgModifyAck	Acknowledge a request for upgrading to exclusive access (is not accompanied by copy of the region data)
MsgModifyAckData	Like MsgModifyAck but is accompanied by copy of the region data

Remote-to-Home Protocol Messages

MESSAGE	DESCRIPTION
MsgSharedReq	Request copy of the region (for shared access)
MsgExclusiveReq	Request copy of the region (for exclusive access)
MsgModifyReq	Request an upgrade to exclusive access to the region
MsgInvalidateAck	Acknowledge an invalidation request (is not accompanied by copy of the region data)
MsgInvalidateAckData	Like MsgInvalidateAck but is accompanied by copy of the region data
MsgRelease	Acknowledge a MsgRInvalidate message by downgrading exclusive access to shared (leaves behind a shared copy of the region)

Figure 5.2 shows the code structure of the synthetic microkernel. Each processor runs a single thread which interacts with other threads through a set of shared objects. Each thread goes through several phases (separated by global barriers) in which it makes a fixed number of accesses (`NOBJS_PER_TRIP`) to the shared objects. The access type and specific object being accessed are a function of the following parameters used to model various execution behaviors:

- *Access distribution* (line 5) models the degree of sharing imbalance in the application and influences the queuing effects of protocol handler occupancy due to spatial irregularity.
- *Read:Write ratio* (line 6) models the sharing pattern and drives the amount of invalidation traffic.

```
1   slave_thread() {
2       // map regions
3       for (i=0; i<NTRIPS; i++) {
4           for (j=0; j<NOBJS_PER_TRIP; j++) {
5               obj = select_object( );
6               op = select_operation( );
7               rgn_start_<op>( obj );
8               // use the object
9               compute( obj );
10              rgn_end_<op>( obj );
11              // do other stuff
12              inter_request_compute( );
13          }
14      }
15      // unmap regions
16  }
```

Figure 5.2 Code structure of the sharing microkernel. The execution behavior of the microkernel can be influenced by varying several parameters (corresponding to the boxed lines): the distribution of shared accesses (line 5), the ratio of read to write operations (line 6), the use granularity (line 9), and the mean and standard deviation of the inter-request granularity (line 12).

- *Use granularity* (line 8) models variances in object size and influences the impact of coherence overheads. The use granularity is used as a scaling factor in the microkernel: all other overheads are expressed relative to it.
- *Mean interrequest granularity* (line 10) models the frequency of shared accesses and influences responsiveness to protocol processing.
- *Standard deviation of interrequest granularity* (line 10) models variance in local computation times and affects responsiveness to protocol processing.

The microkernel is exercised over the range of parameter values listed in Table 5.4. hotspot,×4 and hotspot,×8 refer to skewed object access distributions, where a distinguished node receives four times or eight times, respectively, as many access requests as the other nodes. All experiments reported in this chapter are for 32 processors, and assume an object size of 128 bytes, `NTRIPS` equal to 5, and `NOBJS_PER_TRIP` equal to 1000. To model object reuse, only 10% of all references per processor are to unique objects. Each processor serves as the home node for 10 target objects (except for the contention access distribution patterns discussed below). The access distribution parameter drives selection of the target processor, from which the target object is selected randomly. To keep the amount of sharing per object fixed, the contention access distribution patterns (hotspot,×4, hotspot,×8) assume a proportional increase in target objects on the "hot" nodes.

Table 5.4 Values of sharing microkernel parameters.

PARAMETER	VALUES
Access distribution (d)	random, hotspot×4, hotspot×8
Read:Write ratio (o)	95:5, 90:10, 85:15
Use granularity (τ)	1.0 μs, 10.0 μs, 100.0 μs
Inter-request granularity	
mean (κ)	0.1τ, 1.0τ, 10.0τ
std dev (σ)	0.0, 0.5κ, κ

5.3 Object-consistent DSM: Low-Overhead Coherence

In this section we examine the extent to which sophisticated communication architectures improve the performance of existing object-consistent DSM implementations. Our goal is to determine whether a richer set of communication architecture features can alleviate the performance challenges associated with supporting fine-grained sharing in the context of dynamic computations discussed in Section 5.1.1.

This section is organized as follows. We first describe implementations of the CRL interface on each of the architectural interfaces discussed in Section 3.4. We then present costs of primitive sharing operations for each of these implementations, discussing both qualitative and quantitative differences. The performance of the different implementations is evaluated using the sharing microkernel in Section 5.4.

5.3.1 DSM implementations

In this section, we describe four implementations of the CRL interface, corresponding to the MSG, PUT/GET, LD/ST, and CC-LD/ST communication architecture interfaces. These implementations are shown pictorially in Figure 5.3 and discussed in more detail below.

MSG. The CRL implementation for the MSG interface uses messages which correspond to the protocol events described in Section 5.2. The underlying messaging infrastructure is based upon implementations of the Fast Messages interface described earlier in Section 2.5.3. Message handlers contain

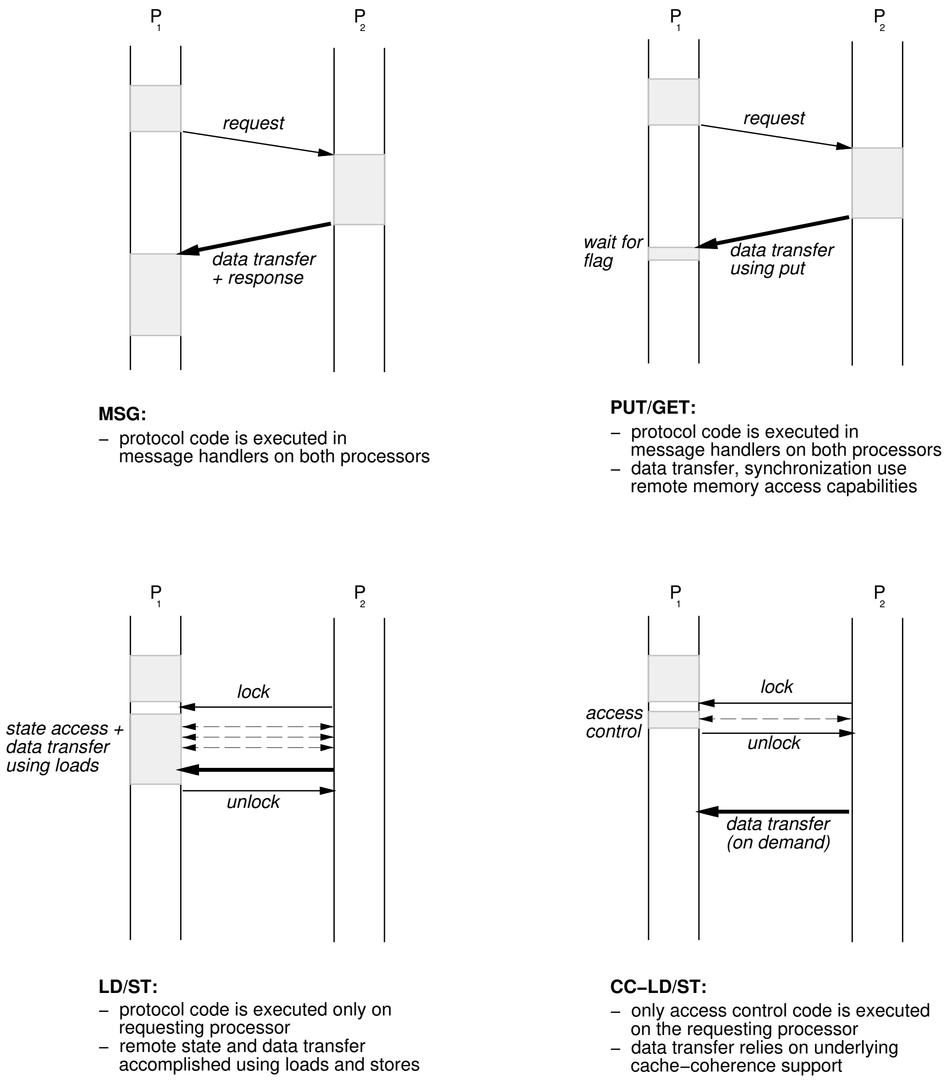

Figure 5.3 Four implementations of the CRL interface corresponding to various communication architectures shown in the context of a read miss. Shaded boxes represent coherence handler, solid lines represent protocol messages, bold lines represent data transfer, and dashed lines represent remote access of protocol state.

the protocol code. Given the emphasis on efficient support for fine-grained software DSM and the large relative costs of interrupts on both our target platforms, explicit polling is used to receive coherence messages. The protocol polls the network on every coherence operation.

PUT/GET. The CRL implementation for the PUT/GET interface retain essentially the same structure as the MSG implementation with the following two exceptions. First, data transfers use the remote memory access capabilities. Second, synchronization between the home node and the requester is achieved via remote stores, thereby reducing synchronization costs. Researchers of page-based DSM systems [102,112,113] have also proposed using remote stores to implement a write-through behavior in the cache. While doing this has a large payoff for page-based DSM systems since it enables support for multiple writers without a complicated diff-based protocol, it makes less of a difference for fine-grained DSM systems. As we show later in the section, the cost of data transfer makes up only a small fraction of the overall miss processing cost.

LD/ST. The CRL implementation for the LD/ST interface executes all protocol code on the requesting node taking out locks as appropriate to update directory state at the home node and to invalidate remote copies. Unlike the MSG and PUT/GET cases, the directory is treated as a distributed structure which can be manipulated by any processor. Protocol events just become function calls. On the T3D, the remote data structures are accessed using uncached loads and locking is implemented using the atomic swap option. On the Origin, the remote data structures are accessed using cached loads[3] and locking is implemented using the hardware fetch-and-op support. As with the PUT/GET implementations, data transfer is implemented using remote memory access capabilities. Protocol invariants are used to minimize the amount of state which needs locking, but the flip side is that even accesses to local regions need to be protected by locks. These implementation have the advantage that they minimize occupancy of nodes other than the requester (requiring their involvement only to the extent required for servicing remote load and lock requests), but this improvement comes at the cost of increased overhead in the protocol code because of accessing nonlocal state.

CC-LD/ST. The CRL implementation on the CC-LD/ST interface builds upon hardware support for cache coherence which maintains the consistency of cache lines at the level of individual load and store operations. Note that implementing the CRL interface requires more than just word-level cache coherence support: the implementation must ensure that all accesses within a `rgn_start_{read,write}`

[3] The hardware ensures that the protocol state is kept consistent.

and `rgn_end_{read,write}` pair complete atomically. The implementation essentially consists of a simple access control protocol for each object that allows exclusive access to writers of the region and shared access to readers. Data transfer relies completely on the underlying cache-coherence support. While an ideal implementation of the CC-LD/ST organization would execute the entire protocol on a dedicated hardware unit (outside the compute processor), we approximate it by moving all but the access control portion of the protocol into hardware.

5.3.2 Primitive operation costs

In this section, we describe the primitive operation costs for object-consistent DSM implementations. Tables 5.5 and 5.6 show the measured costs of primitive sharing operations for each of the implementations on the Cray T3D and the SGI Cray Origin 2000 respectively. We consider the costs of four operations that represent basic protocol operations:

1. Reading an 8-byte region in the HomeExclusive state at the home node. This measures the minimal cost of protocol interaction between the requester and the home node.

2. Reading a 128-byte region in the HomeExclusive state at the home node. When contrasted with the first primitive operation, this measures the additional costs of data transfer in each of the implementations.

3. Writing an 8-byte region currently in the RemoteShared state on another remote node. This measures the additional cost of interacting with a remote node for invalidating a shared copy of the region.

4. Writing an 8-byte region currently in the RemoteShared state on two other remote nodes. This measures the additional per-invalidation cost when more than one copy of the region needs to be invalidated.

The operation costs are organized into three major categories: latency, overhead, and occupancy. *Latency* is the elapsed time between initiation and completion of the operation. *Overhead* is the amount of time spent on the operation by each participating processor and is separated into components at the requester, the directory (home), and the remote nodes. *Occupancy* refers to the portion of the overhead time when the node is unable to process coherence requests meant for *other* objects. The occupancy

Table 5.5 Primitive object sharing operations on the Cray T3D (all costs in μs).

VERSION	LATENCY	OVERHEAD			OCCUPANCY		
		Requester	Directory	Remote	Requester	Directory	Remote
Read clean copy (8 bytes)							
MSG	49.47	24.60	15.59	–	24.60	15.59	–
PUT/GET	26.29	8.72	10.06	–	8.72	10.06	–
LD/ST	28.47	28.47	–	–	0.10	0.10	–
Read clean copy (128 bytes)							
MSG	71.05	37.63	24.14	–	37.63	24.14	–
PUT/GET	28.03	8.72	11.80	–	8.72	11.80	–
LD/ST	32.83	32.83	–	–	0.10	0.10	–
Write with 1 invalidate (8 bytes)							
MSG	92.08	22.17	45.97	12.70	22.17	45.97	12.70
PUT/GET	60.16	5.86	23.80	14.20	5.86	23.80	14.20
LD/ST	74.31	74.31	–	–	0.10	0.10	0.10
Write with 2 invalidates (8 bytes)							
MSG	102.75	22.17	57.82	12.70	22.17	57.82	12.70
PUT/GET	69.97	5.86	33.89	14.20	5.86	33.89	14.20
LD/ST	99.46	99.46	–	–	0.10	0.10	0.10

is typically the same as the overhead when processor participation is needed to complete a coherence action. Thus, occupancy represents the end-point contention effects of coherence processing. As with the overhead, occupancy costs are also split into requester, directory, and remote node components.

The data in the tables show that overall, our implementations of software object shared memory on the four communication architecture interfaces are competitive with the best reported in literature [78, 79, 108]. The following specific observations can be made about the different organizations.

MSG. This implementation incurs the largest latency, overhead, and occupancy, all of which are attributable to the least amount of hardware support. The data shows that the MSG implementation requires participation of the requester and the home node for read operations, and of additional remote processors for the write operations. Thus, processor responsiveness is essential to achieve good performance with the MSG interface.

Comparing across two instances of the MSG interface on the T3D and the Origin, we find that protocol costs on the Origin are a factor of 2-3x better. The relatively high costs on the T3D are due to its small direct-mapped L1 cache and the lack of an L2 cache, while the Origin costs demonstrate the advantages of a more robust cache organization. The difference in costs for reading 128 bytes and

Table 5.6 Primitive object sharing operations on the SGI Cray Origin 2000 (all costs in μs).

VERSION	LATENCY	OVERHEAD			OCCUPANCY		
		Requester	Directory	Remote	Requester	Directory	Remote
Read clean copy (8 bytes)							
MSG	17.12	10.83	10.42	–	10.83	10.42	–
PUT/GET	15.14	6.98	9.14	–	6.98	9.14	–
LD/ST	4.13	4.13	–	–	0.05	0.05	–
CC-LD/ST	1.56	1.56	–	–	0.05	0.05	–
Read clean copy (128 bytes)							
MSG	21.90	13.51	13.37	–	13.51	13.37	–
PUT/GET	16.06	7.6	9.25	–	7.6	9.25	–
LD/ST	4.89	4.89	–	–	0.05	0.05	–
CC-LD/ST	1.56	1.56	–	–	0.05	0.05	–
Write with 1 invalidate (8 bytes)							
MSG	36.19	15.24	17.61	8.68	15.24	17.61	8.68
PUT/GET	31.62	5.99	16.61	10.73	5.99	16.61	10.73
LD/ST	10.07	10.07	–	–	0.05	0.05	0.05
CC-LD/ST	2.24	2.24	–	–	0.05	0.05	–
Write with 2 invalidates (8 bytes)							
MSG	45.88	14.61	26.73	9.76	14.61	26.73	9.76
PUT/GET	41.29	9.98	24.83	10.73	9.98	24.83	10.73
LD/ST	14.76	14.76	–	–	0.05	0.05	0.05
CC-LD/ST	2.43	2.43	–	–	0.05	0.05	–

reading 8 bytes shows the cost of data transfer, which is 21 μs on the T3D, and 4.8 μs on the Origin. The larger costs on the T3D are due to the round-trip latency of the pull-messaging implementation, which is better supported on the Origin.

PUT/GET. This implementation shows the performance advantages of hardware remote memory access capabilities which improve performance over the MSG implementation. As with the MSG implementation, the PUT/GET implementation overlaps the latency associated with multiple invalidations. The CRL implementation on the PUT/GET interface also involves the requester, the home node, and potentially additional remote processors, requiring responsiveness on their part for good performance.

Looking at specific improvements across the two platforms, we find that the PUT/GET implementation reduces operation costs over the MSG implementation by up to 50% on the T3D, and up to 15% on the Origin. On the T3D, the improvement stems from using remote stores for data transfer (thus, the difference is larger for larger messages) and avoiding message-based synchronization between the home node and the requester. On the Origin, the improvement stems from the same reasons but is tempered by the underlying cache-coherent hardware: data ends up in the home node's cache when the latter writes directly into the requester's memory (incurring a round trip in the network), and when the requester finally reads the data, it needs to bring the cache lines back into its own cache (incurring an additional round trip).

LD/ST. The LD/ST measurements show the additional advantages that accrue from remote atomic operations. Although quantitative advantages of the LD/ST interface differ across the two platforms, the measurements show a qualitative difference between the organizations: LD/ST involves only the requester (which executes all the protocol code), in contrast to the MSG or PUT/GET organizations (which require involvement of multiple processors). Thus, performance of the LD/ST organization does not depend upon responsive processors. For the same reason, a processor is tied up with an ongoing coherence action only to the extent that it participates in a remote memory or atomic operation. The LD/ST costs also show a disadvantage: unlike the MSG and PUT/GET organizations, it is not able to overlap processing of multiple invalidations, doing them one after another.

Looking at specific measurements on the two target platforms, we find that on the T3D, the primitive operation costs are actually higher than for the PUT/GET interface: a fact explained by the overheads of accessing remote state in an uncached fashion and the need to protect local protocol state against concurrent access. The Origin costs show that this situation does not exist if cached loads are used, resulting in an improvement of up to 75%. The occupancy of protocol handlers is significantly reduced, equaling only 100 ns on the T3D and 50 ns on the Origin.

CC-LD/ST. Measurements for this implementation show the advantages of shifting most of the protocol actions into hardware. The remaining costs are associated with the access control protocol. As with the LD/ST interface, the CC-LD/ST is less sensitive to processor unresponsiveness due to the minimal occupancy of protocol handlers.

Summary. Implementations of object shared memory for different communication architecture interfaces across our two target platforms exhibit several qualitative and quantitative differences. The overall quantitative trend is that the addition of hardware support (PUT/GET versus MSG, LD/ST versus PUT/GET, and CC-LD/ST versus PUT/GET) results in a reduction in per-operation costs. The notable exception is the LD/ST organization on the T3D which incurs higher costs than the PUT/GET organization. Across the two target platforms, the Origin represents a faster implementation of each architecture. Qualitative differences among the organizations involve sensitivity to processor unresponsiveness and the occupancy due to protocol handler execution. Implementations on the LD/ST and

CC-LD/ST interfaces are less sensitive to unresponsive processors than are MSG and PUT/GET implementations. This is related to the fact that implementations on the LD/ST and CC-LD/ST interfaces incur the least protocol handler occupancy, followed by the PUT/GET interface.

In the next section, we describe how these qualitative and quantitative performance differences translate to overall performance in the context of the sharing microkernel.

5.4 Performance of Object-consistent DSM

In this section, we use the sharing microkernel (see Section 5.2.2) to evaluate the performance of each of the implementations described in the previous section. We describe two sets of experiments, which characterize the performance of various implementation with respect to their occupancy and responsiveness properties. We first present an overview of the experiments, and then discuss each set in more detail.

5.4.1 Overview of experiments

The experiments can be classified into two sets: those that measure the performance sensitivity of an implementation to *responsiveness*, and those that measure the sensitivity to *occupancy*.

Responsiveness experiments. This set measures the extent to which a DSM implementation relies on processor responsiveness for performance. Another perspective on these experiments is that they characterize, for each communication architecture interface, the performance degradation caused due to processor unresponsiveness. This set measures performance for the following two microkernel configurations (the parameter symbols and values are from Table 5.4):

1. The use granularity is varied, keeping the other parameters fixed: d=random, o=95:5, κ=10.0τ, and σ=0.0.

2. The mean interrequest granularity is varied, keeping the other parameters fixed: d=random, o=95:5, τ=10.0 μs, σ=0.0.

Microkernel configurations 1 and 2 model uniform processor unresponsiveness, while configuration 3 captures the temporal irregularity inherent in dynamic multithreaded computations.

Occupancy experiments. This set of experiments measures the extent to which the DSM implementation relies on low processor occupancy for performance. Another perspective on these experiments is that they characterize the queuing (contention) degradation arising from unbalanced sharing. This set measures performance for the following two microkernel configurations:

1. The read-write ratio is varied, keeping the other parameters fixed: d=random, τ=10.0 μs, κ=10.0τ, σ=0.0.

2. The access distribution is varied, keeping the other parameters fixed: o=95:5, τ=10.0 μs, κ=10.0τ, σ=0.0.

The first configuration models an increase in coherence traffic, and the second models the spatial irregularity inherent in dynamic multithreaded computations.

The results of the above experiments on the Cray T3D and the SGI Cray Origin 2000 are discussed in the next two subsections.

5.4.2 Sensitivity to responsiveness

Figure 5.4 shows the normalized microkernel execution time for various communication architecture interfaces on the Cray T3D (left) and the SGI Cray Origin 2000 (right) as the use granularity is varied keeping the other parameters constant as specified above. Figure 5.5 shows the corresponding graphs when the mean interrequest granularity is varied.

The microkernel execution time is broken down into three components: *computation*, *coherence*, and *idle* time. Computation time reflects the sum of the use and inter-request granularities over all iterations of the microkernel. Coherence time reflects overheads of the `rgn_start_{read,write}` and `rgn_end_{read,write}` operations. Idle time reflects the time a processor spends awaiting the completion of a coherence operation. In the LD/ST and CC-LD/ST organizations, these correspond to the time spent spinning on various locks. The graphs show all times normalized with respect to the computation time. In the discussion below, execution overheads refer to the sum of coherence and idle times.

As expected, the execution overheads decrease with increases in use granularity or inter-request compute granularity, both of which decrease the frequency of coherence operations. However, what is interesting is whether this decrease is proportional to the increase in compute granularities and which component of execution overhead dominates as the granularities increase. The graphs show that a

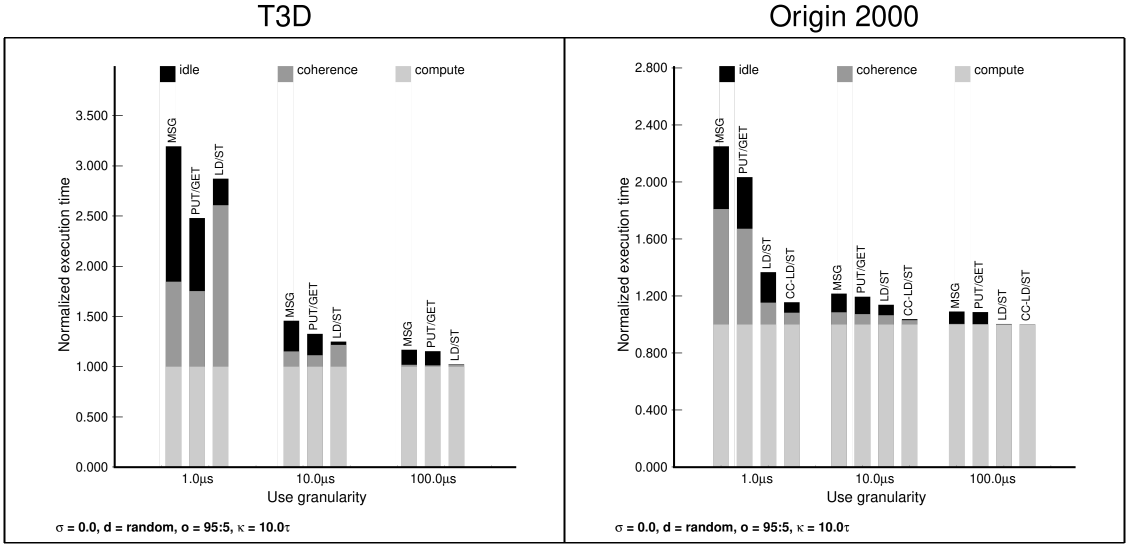

Figure 5.4 Sensitivity to responsiveness: Microkernel execution time breakdown on the T3D and the Origin as the use granularity is varied.

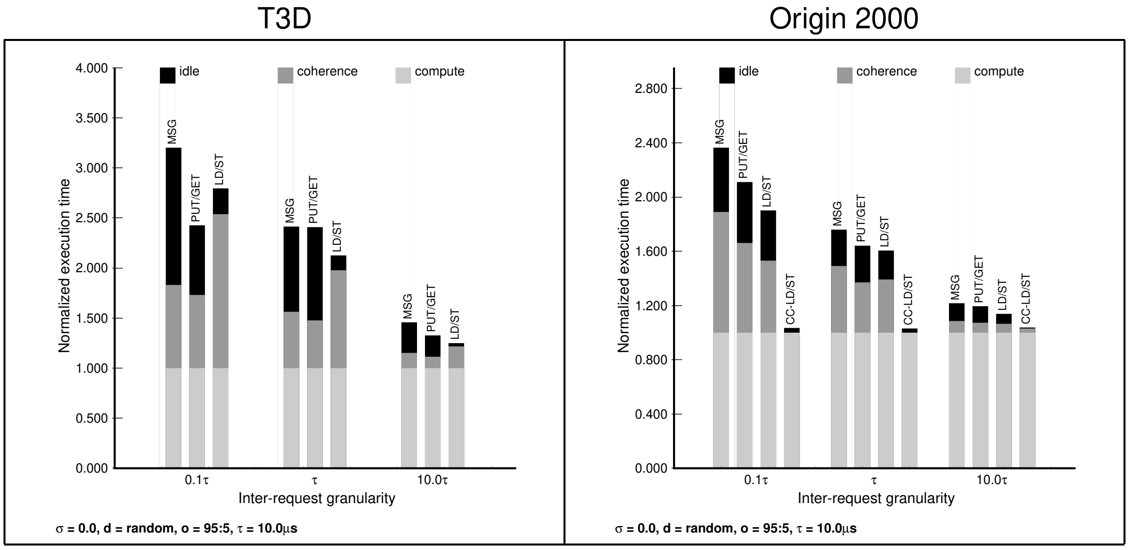

Figure 5.5 Sensitivity to responsiveness: Microkernel execution time breakdown on the T3D and the Origin as the interrequest compute granularity is varied.

particular implementation is sensitive to responsiveness if its overheads do not decrease in proportion to the decrease in coherence operation frequency. The following observations can be made from the graphs for each communication architecture interface:

MSG. For the MSG interface, microkernel execution overheads for larger use or interrequest granularities are dominated by the idle time which does not reduce proportionally. For example, execution overheads only decrease from 2.2 times the computation time to 0.4 times the computation time on the T3D as interrequest granularity increases from 1 μs to 100 μs. A similar observation holds true on the Origin as well. This disproportionate decrease in execution overheads, and the increasing dominance of idle time show that the MSG organization requires responsive processors to deliver good performance.

PUT/GET. As with the MSG interface, microkernel execution overheads for larger use or inter-request granularities are dominated by the idle time which does not reduce proportionally. For example, execution overheads only decrease from 1.4 times the computation time to 0.3 times the computation time on the T3D as inter-request increases from 1 μs to 100 μs. A similar observation holds true on the Origin as well. As earlier, the increasing dominance of idle time shows that the implementation using the PUT/GET interface is also sensitive to processor responsiveness, requiring the latter for performance. The data in the graphs show another interesting point: although the PUT/GET interface results in considerable performance improvement over the MSG interface for smaller use or inter-request granularities, this improvement virtually is non-existent for the larger granularities. Thus, even though the underlying hardware provides low overhead, responsive communication operations, the software protocols are not able to exploit them for a performance advantage.

LD/ST. In contrast to the MSG and PUT/GET interfaces, execution overheads for the LD/ST interface decrease in proportion to the increase in use or inter-request granularity, showing its improved sensitivity to responsiveness. As can be seen from the data in the graphs, the difference between the implementations is the decreased idle time. The reduction in idle time arises from the qualitative nature of the LD/ST implementation which is essentially immune to processor activity on remote processors. In fact on the T3D, this reduction is even able to overcome the performance disadvantages of larger coherence overheads as compared to the MSG and PUT/GET interfaces.

CC-LD/ST. This implementation benefits from moving most of the protocol processing into hardware, and exhibits negligible execution overheads over the entire range of parameters.

5.4.3 Sensitivity to occupancy

Figure 5.6 shows the microkernel execution time for different hardware organizations on the Cray T3D (left) and the SGI Cray Origin 2000 (right) as the read-to-write ratio is varied keeping the other parameters constant as specified earlier. Figure 5.7 shows the corresponding graphs when the shared access distribution is varied. As before, the execution time is broken down into compute, coherence, and idle time components, all normalized with respect to the compute time.

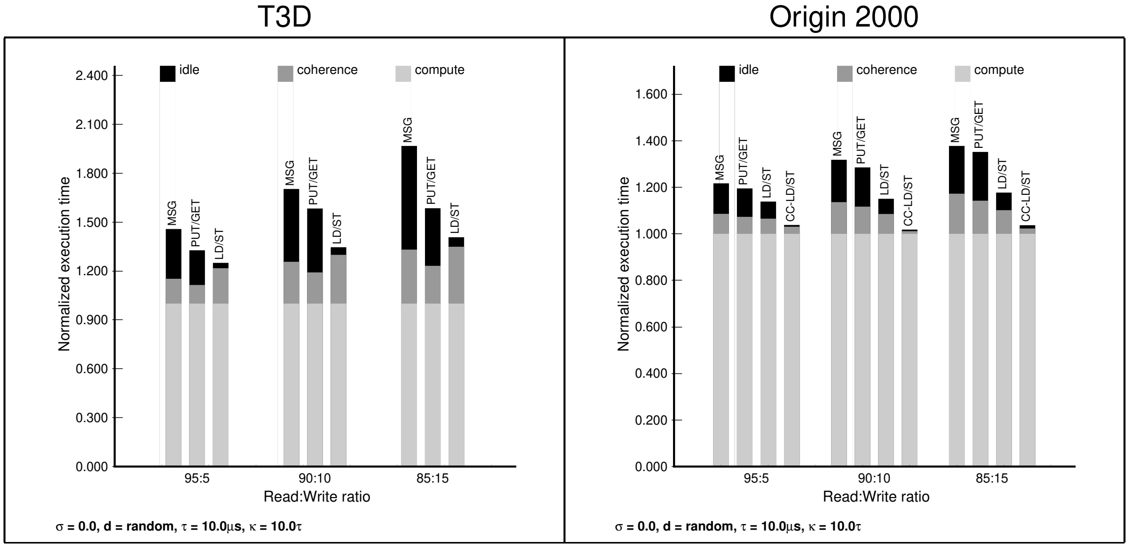

Figure 5.6 Sensitivity to occupancy: Microkernel execution time breakdown on the T3D and the Origin as the read-to-write ratio is varied.

In Figure 5.6, as the read-to-write ratio is decreased, execution overheads increase for all organizations. What is interesting however, is the relative contributions to execution overhead of the idle time and coherence time components. For the MSG and PUT/GET organizations, execution overheads increase due to increases in both idle time and coherence overhead, while in the LD/ST and CC-LD/ST organizations, the increase is mainly due to coherence overheads. For example on the T3D, as read-to-write ratio is decreased from 95:5 to 85:15, execution overhead for the MSG interface increases from 0.4 times the compute time to 0.95 times the compute time (idle time correspondingly increases from 0.3 times to 0.65 times the compute time). For the PUT/GET interface, overhead increases from 0.3 times to 0.6 times the compute time (idle time increases from 0.25 to 0.5 times the compute time). In contrast, the LD/ST interface exhibits an execution overhead increase only from 0.2 times to 0.3 times the compute time, almost all of it attributable to increased coherence overheads. Similar trends are observed

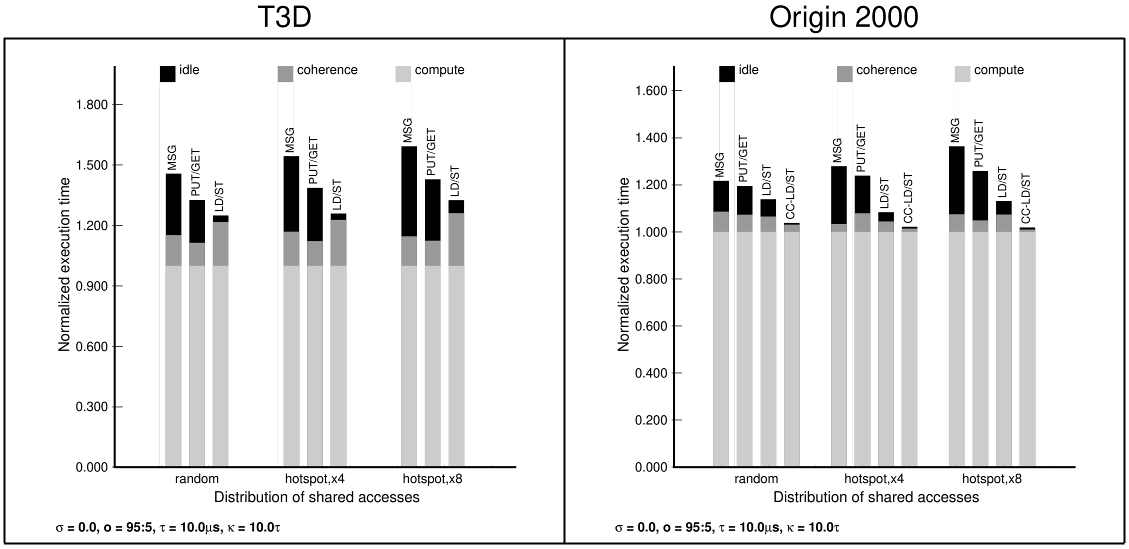

Figure 5.7 Sensitivity to occupancy: Microkernel execution time breakdown on the T3D and the Origin as the distribution of shared accesses is varied.

on the Origin as well. The disproportionate increase in idle time stems from higher protocol handler occupancies of the MSG and PUT/GET organizations which add queuing delays to the processing of incoming coherence messages. As suggested by the per-operation costs in Tables 5.5 and 5.6, idle time increases more for the MSG organization as compared to PUT/GET.

Figure 5.7 shows that the queuing delays are exacerbated when the distribution of shared accesses becomes increasingly skewed. As the access distribution becomes more imbalanced (from random to hotspot,$\times$8), execution overheads of MSG and PUT/GET organizations increase, while those for the LD/ST and CC-LD/ST organizations stay essentially unchanged. The moderate increase in coherence times for all the organizations is attributable to an increase in the time spent by the hotspot node, processing access requests. However, as the figure shows, there is a disproportionate increase in idle time. For example, on the T3D, as access distribution changes from random to hotspot,$\times$8, idle time increases from 0.3 times to 0.65 times the compute time for the MSG implementation, and from 0.2 times to 0.4 times the compute time for the PUT/GET implementation. Thus, these organizations are not robust over imbalanced sharing, unlike LD/ST and CC-NUMA which benefit from lower protocol handler occupancies.

5.4.4 Summary

In this section, we have shown in the context of a sharing microkernel that existing object-consistent DSM protocols exhibit different sensitivity to responsiveness and occupancy which varies as a function of the underlying communication architecture interface.

The MSG and PUT/GET interfaces show increased performance sensitivity to responsiveness and occupancy, resulting in increased idle time as processors become unresponsive or busy processing lots of coherence requests. In contrast, the LD/ST and CC-LD/ST interfaces are less sensitive to responsiveness and occupancy, delivering performance essentially unaffected by them. The performance improvement arises from the availability of remote atomic operations and memory access operations in these architectures which permits coherence protocols to be implemented completely on the requesting processor. On the T3D, the resulting idle time reductions in the LD/ST interface can actually compensate for its higher per-operation coherence overheads as compared to the MSG and PUT/GET interfaces.

Our microkernel results suggest that the reduced sensitivity of the LD/ST interface to responsiveness and occupancy may be crucial for obtaining good performance for dynamic computations using existing DSM protocols. We corroborate this conclusion using application programs in Chapter 7. Our results also show that the mere availability of low overhead, responsive communication operations as in the PUT/GET interface is not sufficient to obtain robust sharing performance. Existing DSM protocols are not able to exploit these characteristics for a performance advantage.

5.5 View Caching: Latency-Tolerant Coherence

In this section we describe *view caching*, a framework for the construction of latency-tolerant custom coherence protocols which require less synchronization among the participating processors, and hence are more tolerant to unresponsive processors and unbalanced sharing. In addition, unlike existing DSM protocols, our solution leverages the availability of responsive communication architecture features to implement these custom protocols efficiently, yielding robust sharing performance.

The rest of the section is organized as follows. Section 5.5.1 revisits the use of synchronous (request-reply) protocols in existing DSM systems, and Section 5.5.2 introduces our solution. Section 5.5.3 describes the view caching framework in more detail. We conclude in Section 5.5.4 by discussing the implementation of the framework in the context of the object-consistent DSM system described earlier.

5.5.1 Rationale: Synchronous DSM protocols

Existing DSM techniques utilize synchronous (request-reply) protocols because they adopt a memory controller perspective of coherence. Coherence actions are determined by a single parameter, the consistency model, which reflects the lowest common denominator of application consistency requirements. However, consistency models only specify two aspects of coherence — the coherence granularity, and when data needs to be made consistent — leaving it to the DSM implementation to determine *what it means for copies to be consistent* and *how coherence is actually implemented*. Given the memory controller perspective, coherence is defined only in terms of the observable *effect* of accesses on the *state* of a shared object, i.e., at the level of loads and stores to memory locations. Consequently, most implementations conservatively define object copies to be consistent only if they reflect identical state. As for the problem of implementing coherence, most protocols[4]

- maintain globally accessible information (the *directory*) about object state and its copies (i.e., where it resides and the access privileges of its copies),
- query this information prior to satisfying an access request, and
- conservatively invalidate conflicting copies before granting access.

Querying directory information requires a request-reply protocol, coupling activities on the requesting processor and the processor storing the information. Similarly, invalidating conflicting copies requires synchronous protocols that couple activities across the requesting processor, the processor initiating the invalidates, and the processor holding the copy which needs to be invalidated.

The underlying cause for this synchronization and coupling is the fact that coherence actions are initiated at the level of accesses to memory state *without utilizing any higher-level semantic information about the access*. While such coupling does not degrade performance for hardware cache coherence protocols with responsive cache controllers, this is not the case for a software DSM system. Coupling activities on processors that are unresponsive to coherence messages degrades performance due to an increase in processor idle times.

[4] Our discussion here focuses on invalidation-based directory protocols, although similar arguments hold true for other protocols as well.

5.5.2 Solution overview

Our solution for latency-tolerant coherence protocols exploits application knowledge to supply information that is not provided by the consistency model, specifically what it means for object copies to be consistent, and how object coherence is implemented. This approach is motivated by two observations:

1. Exploiting different consistency requirements for different program objects can reduce coherence traffic by eliminating the need for some coherence actions.

2. Exploiting higher-level consistency semantics (based on application use) can allow the optimized implementation of remaining protocol actions.

A natural unit of shared object use (in a concurrent object-oriented programming context) is a user-defined method. We define an object "view" to be the object state accessed by a method and consider it as the coherence unit. Application knowledge about data-access semantics as well as derived knowledge about object copies is considered at this level. This view-specific knowledge is used to construct customized coherence protocols for object caching (hence the name, *view caching*). Two kinds of application knowledge are useful for customizing protocols:

- *Local knowledge* is information local to the method which can be derived automatically by a compiler.

- *Global knowledge* is information about global consistency requirements and concurrent program activities.

Examples of local knowledge include the subset of object state that the method accesses, its side-effects (e.g., accumulation of a variable), and its interaction with executions of other methods (e.g., commutativity). Examples of global knowledge include knowing that threads can compute with out-of-date values of object state, information about program synchronizations, and the effect of concurrently-active threads on shared state.

The above kinds of application knowledge relax the consistency requirements (as compared to object consistency) by dictating the required visibility of updates, and the semantics of view consistency. In addition, such information allows global information about the states of object copies to be derived, enabling construction of asynchronous, optimized protocols. For instance, the knowledge that a view is never modified past a program synchronization point allows one to derive the information that (1) the copy of the view at the home node is up-to-date, and (2) all object views acquired since that time have read-only access. This information simplifies the coherence actions required to obtain a new copy of the object view.

5.5.3 View caching framework

Coherence protocols include protocol elements on the directory (which stores location information of object copies) and the cache (which maps global names to local copies), and data transfer. Figure 5.8 shows these components in the context of a write miss. The shaded rectangles correspond to protocol code executed on the participating nodes, and the arrows represent messages sent between them. A write miss at the requester causes a request to be sent to the home node. The home node receives the request, and because there is a shared copy outstanding, it sends an invalidation request to a remote node. The remote node receives the invalidation request and acknowledges it, either immediately or when it is safe to revoke access. The acknowledgment arrives at the home node which then changes the directory state to reflect the new owner and then transfers the object state to the original requester.

While most DSM systems bundle these components into a single general-purpose protocol that is applied for all accesses, view caching distinguishes among three components that are shown as outlined regions in Figure 5.8:

1. `access-grant` consists of processing (at the cache and directory) to satisfy the requested access, including revoking access that was previously granted.
2. `access-revoke` disables further use of the copy.
3. `data-transfer` transfers object state between processors.

View caching coherence protocols are built by assembling customized implementations of these components. Knowledge of application data access semantics drives this customization, focusing particularly on decreasing message traffic and required synchronization. As we shall see later, reducing synchronization is key to leveraging the low-overhead responsive communication mechanisms that constitute the communication architectures discussed earlier.

To keep protocol variety manageable, our framework selects from a predefined set of component implementations. As we shall see in Chapter 7, view caching achieves good performance despite this

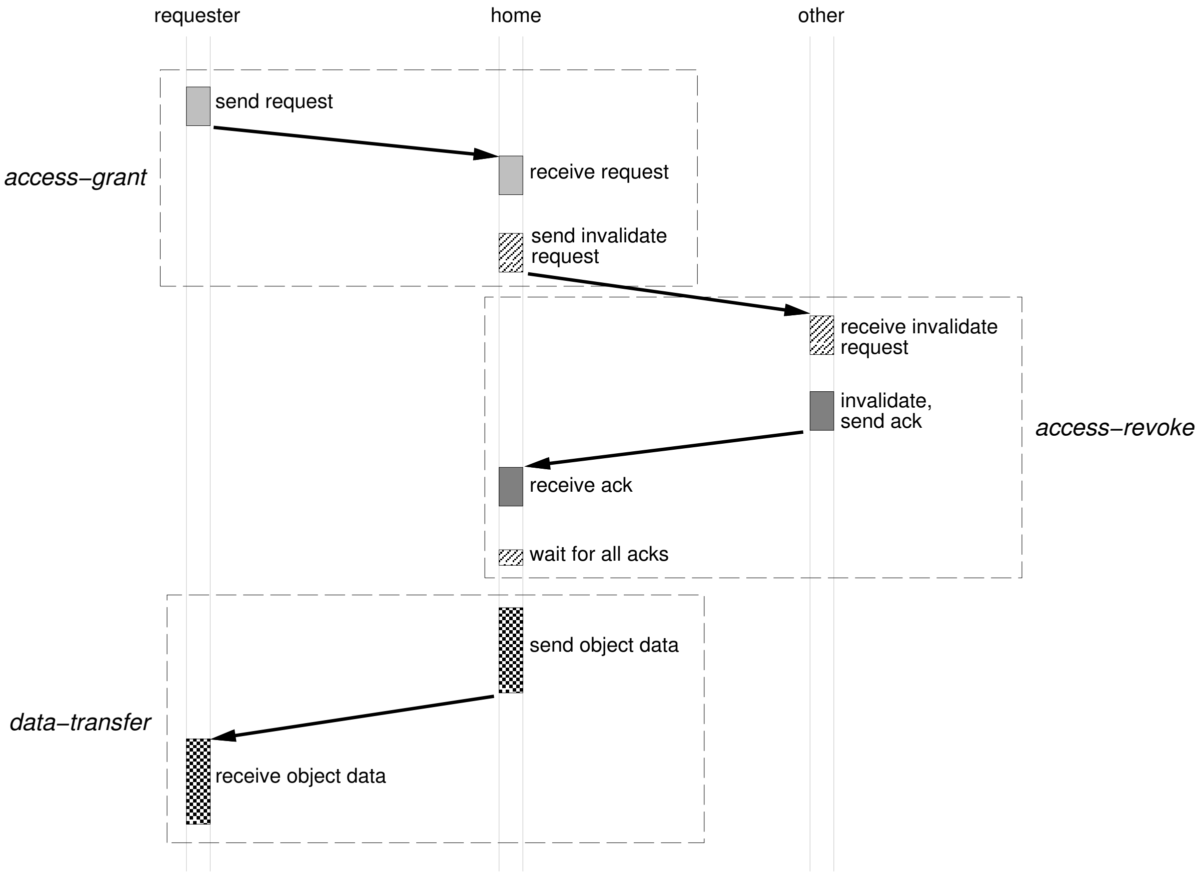

Figure 5.8 Components of cache processing for a write miss. Shaded rectangles represent protocol code executed on the participating nodes.

limited flexibility. The predefined component implementations incur less overhead and require reduced synchronization. The selection process is driven by specifying values for a set of parameters which are chosen to correspond naturally to different decision points in the implementation of a component.

The customizable parameters for each component and their allowed values are listed in Table 5.7, and described below.

access-grant. Three parameters customize the implementation of this component: decision?, invalidate?, and dir_update?. The decision? parameter controls where the access-granting decision is performed. It can be performed on either the requester (if the directory state can be inferred using application knowledge), or the home node (when the decision requires access to the directory state stored there). A message is required only if the decision is made on the home node. The invalidate? parameter controls the conditions under which existing copies are invalidated prior to satisfying an

Table 5.7 Options and values for customizing different components of coherence processing.

`access-grant`

PARAMETER	DESCRIPTION	VALUES
decision?	who makes the access decision?	requester / home
invalidate?	when are existing copies invalidated?	never / always / conflicts
dir_update?	how is directory state updated?	no update / asynchronous / synchronous

`access-revoke`

PARAMETER	DESCRIPTION	VALUES
initiator?	who initiates the invalidation?	home / self
ack_reqd?	are invalidations acknowledged?	yes / no

`data-transfer`

PARAMETER	DESCRIPTION	VALUES
initiator?	who initiates the data transfer?	home / requester

access request. Invalidation can be unnecessary (when it can be inferred that the current access does not conflict with any existing copy), always required (when it can be inferred that current access does conflict with all existing copies), or required only on conflicts (e.g., a read access requiring invalidation of existing writable copies and vice versa). The dir_update? parameter controls whether the current directory state needs to be updated as part of granting an access request. Either no update is required (if all accesses to the object can infer its current state), or an update is required. In the latter case an acknowledgment may or may not be required, permitting the use of asynchronous protocols.

access-revoke. This component is customized using two parameters: initiator? and ack_reqd?. The initiator? parameter determines who invokes previously granted access to an object copy. Access to a cached copy may be revoked either voluntarily (e.g., with the requester determining that it has no further need for the copy) or on demand by the home node. In the former case, no invalidation messages are required. The ack_reqd? parameter controls whether an acknowledgment is required when access is revoked by the home node. If no acknowledgment is required, an asynchronous protocol can be used.

data-transfer. This component is customized in terms of one parameter: initiator?. The parameter determines whether the data transfer can be initiated by the requester (when it can be inferred that the copy at the home node has the desired state), or needs the home node's involvement. The former case is optimized by the exploitation of one-sided communication hardware.

We integrate the custom view caching protocols with the CRL interface as shown in Table 5.8, adding a parallel set of calls which work with object views. The `rgn_create_viewable` function creates a region which supports the caching of some of its views. It takes as arguments the number of object views and a two-dimensional array describing conflicts between view pairs. Prior to using a custom protocol to access an object view, the object must be mapped into the local address space using `rgn_map_view`. The latter takes as arguments the custom protocol (which is an index into a library of predefined custom protocols) and the view identifier (which uniquely identifies the object subset that constitutes the view). Finally, object access must be prefaced with `rgn_start_{read,write}_view` and `rgn_end_{read,write}_view` calls, all of which take the view identifier as an argument. In the next section, we describe our prototype implementation of view caching protocols that support this interface.

Table 5.8 View caching extensions to the region-based DSM interface.

OPERATION	FUNCTION
Allocation	`rid_t rgn_create_viewable(int size, void *obj, int nviews, int **conflict_map)` Sets up metadata structure to enable caching of up to `nviews` of the region with view conflicts described by `conflict_map`
Renaming	`void* rgn_map_view(rid_t id, int otype, int vtype, int vid)` Map a view of type `otype` associated with region identifier `id` into the local address space. The view identifier is `vid` and its custom coherence protocol is specified by `vtype`
Access	`void rgn_start_read_view(void *addr, int vid)` Initiate a read view operation on a region
	`void rgn_end_read_view(void *addr, int vid)` Terminate a read view operation on a region
	`void rgn_start_write_view(void *addr, int vid)` Initiate a write view operation on a region
	`void rgn_end_write_view(void *addr, int vid)` Terminate a write view operation on a region

5.5.4 Prototype view caching implementation

There are two aspects of the implementation: a library of view coherence types, and implementation internals such as view caching protocol states and events. These aspects are discussed in more detail below.

5.5.4.1 Library of view coherence types

For our prototype implementation, we instantiate the view caching framework to implement a suite of six custom coherence types. Of these, four are for reading views (`READ_SYNC`, `READ_ASYNC`, `READ_SELF_INV`, and `READ_CONSTANT`), and two for writing (`WRITE_SYNC` and `WRITE_ALONE`). The selection of these six protocols, which exercise all the customization parameters of the three coherence components, was guided by the requirements of applications described in Chapter 7.

Table 5.9 lists the customization of coherence protocol components for these six custom coherence types. The six types span three categories of views, distinguished by how their consistency is coupled with the state of the home node copy. A view can be *synchronous*, *asynchronous*, or *detached*. Synchronous views are tightly coupled with the home node copy: all object views have state identical to that at the home node. Asynchronous views are loosely coupled: object views differ for a period of time but eventually become consistent. Finally, detached views have no coupling with the copy at the home node: they are essentially a snapshot of the current state there. The view coherence types are described in detail below.

1. `READ_SYNC` represents the baseline read view which improves over reading the entire object by eliminating false sharing. The coherence protocol obtains a synchronous view, making the `access-grant` decision on the home node while invalidating any conflicting copies via a synchronous protocol. Data transfer is home-initiated once it ascertains that the current object copy has up-to-date state.

2. `READ_ASYNC` utilizes application knowledge that the accessed object state subset is currently consistent and either is not going to be concurrently updated, or that out-of-date values are permissible. This allows the protocol to infer that the current access does not conflict with any existing copies, and that invalidation requests for the current copy will not appear until the end of the phase. This enables a custom protocol for obtaining an asynchronous view which resolves `access-grant` on the requester, asynchronously records the requester as a reader to permit subsequent invalidation (if required), and relies on requester-initiated `data-transfer`.

Table 5.9 Library of view coherence types, their description, and the corresponding coherence protocol components.

VIEW	DESCRIPTION	COHERENCE PROTOCOL COMPONENTS					
		access-grant			access-revoke		data-transfer
		decision?	invalidate?	dir_update?	initiator?	ack_reqd?	initiator?
READ_SYNC	synchronous view (shared access)	home	conflicts	synchronous	home	yes	home
READ_ASYNC	asynchronous view (shared access) utilizes application knowledge that home node contains valid data, and view is not concurrently updated or out-of-date values are permissible	requester	never	asynchronous	home	no	requester
READ_SELF_INV	detached view (self-invalidated when appropriate) utilizes application knowledge that home node contains valid data, and view is not updated or out-of-date values are permissible until self-invalidation	requester	never	no update	self	no	requester
READ_CONSTANT	detached view (never invalidated) utilizes application knowledge that home node contains valid data which never changes	requester	never	no update	–	–	requester
WRITE_SYNC	synchronous view (exclusive access)	home	conflicts	synchronous	home	yes	home
WRITE_ALONE	detached view (flushed when appropriate) utilizes application knowledge that home node contains valid data, and there are no concurrent readers until after flush	requester	never	no update	self	no	requester

3. `READ_SELF_INV` utilizes application knowledge that the accessed object state subset is currently consistent and will remain so until the next synchronization point. This allows the protocol to infer that the current access does not conflict with any existing copies, and that invalidation requests will definitely appear at the end of the phase. This enables a custom protocol for obtaining a detached view, which relies on requester-initiated `access-grant` and `data-transfer`, requiring home node participation only for data transfer as long as each processor invalidates all cached copies at the next synchronization point.

4. `READ_CONSTANT` utilizes application knowledge that the accessed object state subset is currently consistent and will continue to remain as such. This allows the protocol to infer that the current access does not conflict with any existing copies, and that no invalidation requests will be forthcoming. This enables a custom protocol for obtaining a detached view which requires home node participation only for data transfer.

5. `WRITE_SYNC` represents the baseline write view which improves over writing the entire object by eliminating false sharing. The coherence protocol obtains a synchronous view, making the `access-grant` decision on the home node while invalidating any conflicting copies via a synchronous protocol. Data transfer is home-initiated once it ascertains that the current object copy up-to-date state.

6. `WRITE_ALONE` utilizes application knowledge that the accessed object state subset is currently consistent and not being read concurrently by anyone. This allows the protocol to infer that the current access does not conflict with any existing copies, enabling a custom protocol for obtaining a detached view which requires home node involvement only for data transfer as long as the requester flushes its modifications to the home node upon completion.

The above protocols utilize different communication architectures in identical fashion to the base CRL protocol described in Section 5.2. In addition, the `data-transfer` component in four of the protocols (`READ_ASYNC`, `READ_SELF_INV`, `READ_CONSTANT`, and `WRITE_ALONE`) protocols can take advantage of the PUT/GET interface, completing data transfer without involving the home node. Also, the invalidation requests sent by the `READ_ASYNC` protocol do not require acknowledgments: an asynchronous protocol suffices because reader information is required only after a program synchronization.

The implication of reduced home node involvement, asynchronous protocols, and one-sided communication operations is that view caching protocols incur reduced performance degradation due to poor responsiveness and large protocol handler occupancies.

5.5.4.2 Implementation internals

The above protocols are implemented by extending the basic CRL state machines on the home and remote nodes with the states and events shown in Tables 5.10 and 5.11, respectively. While no new home-side protocol states are required, implementation of the view caching protocols requires augmenting the remote-side state machine with 12 additional states. These 12 states are organized into three groups—RemoteView{...}, RemoteAsyncView{...}, and RemoteDetachedView{...}—that correspond to synchronous, asynchronous, and detached views, respectively. Each group comprises four states representing the situation where the object copy is invalid, where it is valid, where a request to obtain a valid copy is pending, and where a read is in progress.

Events initiated by the remote node include requests for obtaining detached or asynchronous views (MsgSnapshotReq, MsgSnapshotRegisterConsumerReq, and MsgRegisterConsumerReq), and for view invalidation requests and acknowledgments. Synchronous views are obtained using access requests identical to the original CRL protocol. The MsgInvalidateViewReq message is used to explicitly request invalidation of conflicting asynchronous views prior to permitting a write view access to proceed.[5] Messages from the home node to the remote node send data in response to requests for detached and asynchronous views (MsgSnapshotAckData), and acknowledge invalidation requests (MsgInvalidateViewAck) from synchronous write views.

These additional states and events map to different hardware organizations in similar fashion to the base CRL protocols described in Section 5.3. One notable exception is that detached and asynchronous views enable the use of requester-initiated data transfer (using gets in the PUT/GET organization, and remote loads in the LD/ST organization). Requester-initiated `access-revoke` (i.e., self invalidation) is implemented by maintaining, for each synchronization point, a list of views that need invalidating.

The next section describes the primitive operation costs for implementations of two representative view coherence types on different hardware organizations.

[5] Asynchronous read views can be obtained even when a write view operation is in progress (particularly when application knowledge permits the use of stale data). To bound the staleness, we have chosen to invalidate such views on every write.

Table 5.10 Summary of view caching protocol states.

Remote-side Protocol States

STATE	DESCRIPTION
RemoteViewInvalid	This node does not have a valid synchronous view of the region
RemoteViewInvalidReq	Like RemoteViewInvalid, but a request to acquire a valid synchronous view is pending
RemoteViewValid	This node has a valid synchronous view of the region (with shared access)
RemoteViewValidRip	Like RemoteViewValid, plus one or more read operations are in progress
RemoteAsyncViewInvalid	This node does not have a valid asynchronous view of the region
RemoteAsyncViewInvalidReq	Like RemoteAsyncViewInvalid, but a request to acquire a valid asynchronous view is pending
RemoteAsyncViewValid	This node has a valid asynchronous view of the region (with shared access)
RemoteAsyncViewValidRip	Like RemoteAsyncViewValid, plus one or more read operations are in progress
RemoteDetachedViewInvalid	This node does not have a valid detached view of the region
RemoteDetachedViewInvalidReq	Like RemoteDetachedViewInvalid, but a request to acquire a valid detached view of the region is pending
RemoteDetachedViewValid	This node has a valid detached view of the region
RemoteDetachedViewValidRip	Like RemoteDetachedViewValid, plus one or more read operations are in progress

Table 5.11 Summary of view caching protocol events.

Home-to-Remote Protocol Messages

MESSAGE	DESCRIPTION
MsgSnapshotAckData	Acknowledge a request for a snapshot of the region
MsgInvalidateViewAck	Acknowledge a request to invalidate conflicting views

Remote-to-Home Protocol Messages

MESSAGE	DESCRIPTION
MsgSnapshotReq	Request snapshot of the region
MsgSnapshotRegisterConsumerReq	Request snapshot of the region and register consumer (for shared access)
MsgRegisterConsumerReq	Request registration of consumer (for shared access)
MsgInvalidateViewReq	Request invalidation of conflicting views
MsgInvalidateViewAck	Acknowledge a view invalidation request

5.5.5 Primitive operation costs

We consider implementations of an asynchronous (READ_ASYNC) and a detached view coherence type (READ_SELF_INV) on the four communication architectures described earlier. Tables 5.12 and 5.13 show the costs incurred by these implementations on the Cray T3D and the SGI Cray Origin 2000, respectively. We restrict our attention to the read operations since they benefit most from the custom view coherence protocols. In the tables, implementations of the READ_ASYNC and READ_SELF_INV protocols are labeled by attaching the suffixes -ASYNC and -SELF, respectively after the name of the communication architecture interface. For example, implementations of these protocols on the MSG interface are called MSG-ASYNC and MSG-SELF, respectively. As in Section 5.3.2, the tables show the cost breakdown into latency, overhead, and occupancy components.

Table 5.12 Primitive object sharing operations on the Cray T3D corresponding to two view coherence types (all costs in μs).

VERSION	LATENCY	OVERHEAD			OCCUPANCY		
		Requester	Directory	Remote	Requester	Directory	Remote
Read clean copy (8 bytes)							
MSG	49.47	24.60	15.59	–	24.60	15.59	–
MSG-ASYNC	59.75	28.76	24.39	–	28.76	24.39	–
MSG-SELF	51.65	26.32	16.26	–	26.32	16.26	–
PUT/GET	26.29	8.72	10.06	–	8.72	10.06	–
PUT/GET-ASYNC	34.63	29.91	7.22	–	29.91	7.22	–
PUT/GET-SELF	19.62	19.62	–	–	19.62	–	–
LD/ST	28.47	28.47	–	–	0.10	0.10	–
LD/ST-ASYNC	30.51	30.51	–	–	0.10	0.10	–
LD/ST-SELF	20.44	20.44	–	–	0.10	0.10	–
Read clean copy (128 bytes)							
MSG	71.05	37.63	24.14	–	37.63	24.14	–
MSG-ASYNC	81.33	41.79	32.94	–	41.79	32.94	–
MSG-SELF	73.23	39.35	24.81	–	39.35	24.81	–
PUT/GET	28.03	8.72	11.80	–	8.72	11.80	–
PUT/GET-ASYNC	38.99	34.27	7.22	–	34.27	7.22	–
PUT/GET-SELF	23.98	23.98	–	–	23.98	–	–
LD/ST	32.83	32.83	–	–	0.10	0.10	–
LD/ST-ASYNC	34.87	34.87	–	–	0.10	0.10	–
LD/ST-SELF	24.80	24.80	–	–	0.10	–	–

Table 5.13 Primitive object sharing operations on the SGI Cray Origin 2000 corresponding to two view coherence types (all costs in μs).

VERSION	LATENCY	OVERHEAD			OCCUPANCY		
		Requester	Directory	Remote	Requester	Directory	Remote
Read clean copy (8 bytes)							
MSG	17.12	10.83	10.42	–	10.83	10.42	–
MSG-ASYNC	16.43	10.41	9.53	–	10.41	9.53	–
MSG-SELF	16.71	10.96	10.72	–	10.96	10.72	–
PUT/GET	15.14	6.98	9.14	–	6.98	9.14	–
PUT/GET-ASYNC	3.23	3.01	0.09	–	3.01	0.09	–
PUT/GET-SELF	3.18	3.14	–	–	3.14	–	–
LD/ST	4.13	4.13	–	–	0.05	0.05	–
LD/ST-ASYNC	2.88	2.88	–	–	0.05	0.05	–
LD/ST-SELF	2.82	2.82	–	–	0.05	–	–
CC-NUMA	1.56	1.56	–	–	0.05	0.05	–
Read clean copy (128 bytes)							
MSG	21.90	13.51	13.37	–	13.51	13.37	–
MSG-ASYNC	21.21	13.09	12.48	–	13.09	12.48	–
MSG-SELF	21.49	13.64	13.67	–	13.64	13.67	–
PUT/GET	16.06	7.6	9.25	–	7.6	9.25	–
PUT/GET-ASYNC	3.99	3.77	0.09	–	3.77	0.09	–
PUT/GET-SELF	3.94	3.84	–	–	3.84	–	–
LD/ST	4.89	4.89	–	–	0.05	0.05	–
LD/ST-ASYNC	3.64	3.64	–	–	0.05	0.05	–
LD/ST-SELF	3.56	3.56	–	–	0.05	–	–
CC-NUMA	1.56	1.56	–	–	0.05	0.05	–

`READ_ASYNC`. The costs of these implementations show the advantages and disadvantages of using requester-initiated data transfer and asynchronous protocols. These protocol differences result in a cost difference only on the PUT/GET interface with different characteristics for the two target platforms. The relative high costs of remote loads (versus stores) on the T3D result in a net increase in the per-operation cost. In contrast, on the SGI, requester-initiated data transfer requires only one round-trip network protocol to move data from the home to the requester (versus two when the data transfer is home-initiated). This improvement results in a per-operation cost reduction of 75%. Implementations on the MSG and LD/ST interfaces result in the same number of messages as the base protocols, with only minor book-keeping differences in protocol code.

However, the primitive costs do not show an important qualitative difference between the base and READ_ASYNC protocols: the former only requires participation of the home node, even when some other processor holds a modifiable copy of the object. Consequently, READ_ASYNC protocols are expected to show reduced sensitivity to unresponsiveness on remote (non home) processors.

READ_SELF_INV. The costs of these implementations reflect the advantages of requester-initiated data transfer and reduced involvement of the home node. The most noticeable cost differences are for the PUT/GET and LD/ST interfaces where the custom protocol allows the read miss to be serviced without any home node involvement. On the T3D, the relative high costs of remote loads restrict the per-operation cost improvement to 20%. On the Origin, a reduction in network round-trips reduces costs by over 75%. In addition, the table implicitly shows that the -SELF protocol is essentially immune to home node activity. Protocol handler occupancy is an issue only on the requester, implicitly showing that the READ_SELF_INV protocol is essentially isolated from remote processor activity. On the other hand for the the MSG interface, home node involvement is still required for data transfer. Costs are comparable to the base protocol modulo minor book-keeping differences.

As with the costs of the READ_ASYNC implementations, the tables do not show a qualitative advantage of the READ_SELF_INV protocol: on the MSG interface, the latter only requires participation of the home node, even when some other processor holds a modifiable copy.

In summary, view coherence protocols result in quantitative per-operation cost advantages only on some of the communication architectures. However, as we shall show in the next section, their asynchronous nature and reduced synchronization requirements contribute to a bigger performance gain, often overcoming any per-operation cost disadvantages.

5.6 Performance of View Caching

In this section, we evaluate the performance of view caching protocols using the synthetic sharing microkernel presented in Section 5.2.2 to characterize the performance of two view coherence types — READ_ASYNC, and READ_SELF_INV.

The sharing microkernel is converted to use view caching protocols by replacing the calls in lines 7 and 10 to `rgn_start_<op>` and `rgn_end_<op>` with the corresponding view caching variants. All read accesses are altered to use either the READ_ASYNC or READ_SELF_INV protocols. All write accesses are assumed to use the WRITE_SYNC protocol. For brevity, in the following graphs and discussion, microkernel implementations corresponding to the base object consistency, READ_ASYNC and READ_SELF_INV protocols are referred to by the labels BASE, ASYNC, and SELF, respectively.

As in Section 5.4, we report on two sets of experiments which characterize the sensitivity to responsiveness and occupancy, respectively. The microkernel configurations used in the following experiments are identical to those discussed in Section 5.4.1.

5.6.1 Sensitivity to responsiveness

Figure 5.9 shows microkernel execution times for implementations of the two view coherence protocols on various communication architectures as the use granularity is varied, keeping other parameters constant as specified in Section 5.4.1. Figure 5.10 shows the corresponding graphs for variations in the mean interrequest granularity. Each figure consists of six graphs: the left graphs correspond to the Cray T3D while the right graphs represent implementations on the SGI Origin 2000. For each target platform, the three graphs correspond to implementations on the MSG, PUT/GET, and LD/ST architectures. The microkernel execution times of the base protocols (without view caching) are included in each graph for comparison. As in Section 5.4.1, the execution time is broken down into compute, coherence, and idle time components, all normalized with respect to the compute time.

The overall trend across the graphs is that, for each communication architecture, execution overheads decrease with use of view coherence protocols. For the ASYNC protocols, the improvement over BASE is attributable to a reduction in idle time, stemming from the use of asynchronous protocols. For the SELF protocols, the improvement over BASE and ASYNC is due to a reduction in both coherence overhead and idle time. Both reductions stems primarily from a reduced volume of coherence operations, enabled by application semantics that allow use of stale data from the previous iteration.

Within the above general trend, implementations of the protocols on various communication architectures and different target platforms exhibit some quantitative differences. A closer look at these differences provides insight into how each protocol operates. Below, we discuss the key performance differences for each communication architecture.

MSG. ASYNC protocols produce small improvements over the BASE protocol, more so on the Origin than on the T3D. While the improvements results from a reduction in idle time, their magnitude is limited by the fact that home node involvement is still required for coherence processing. In contrast, the SELF protocols reduce execution overheads by a considerable amount, resulting from decreases in

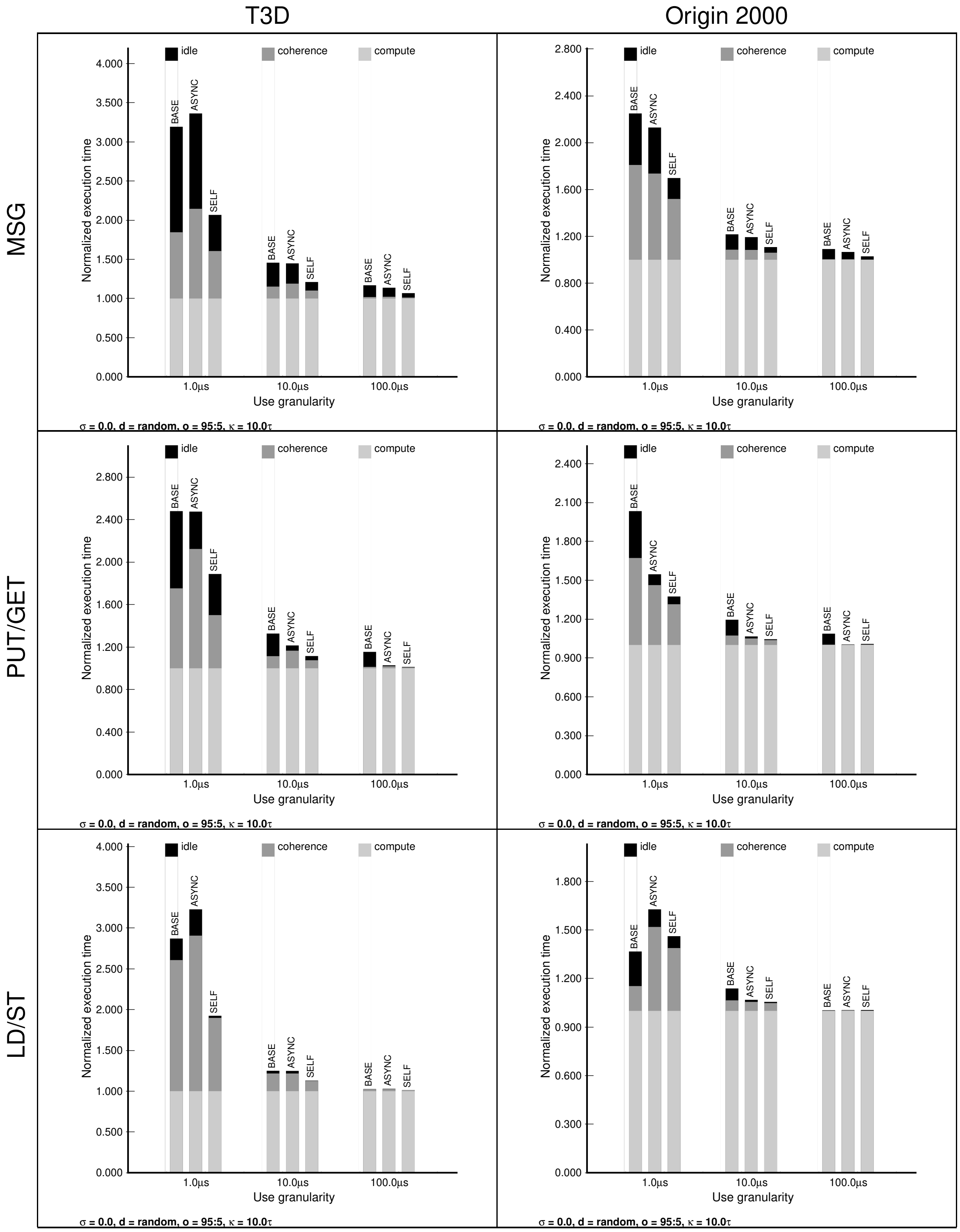

Figure 5.9 Sensitivity to responsiveness: Microkernel execution time breakdown on the T3D and the Origin showing impact of view caching protocols as use granularity is varied.

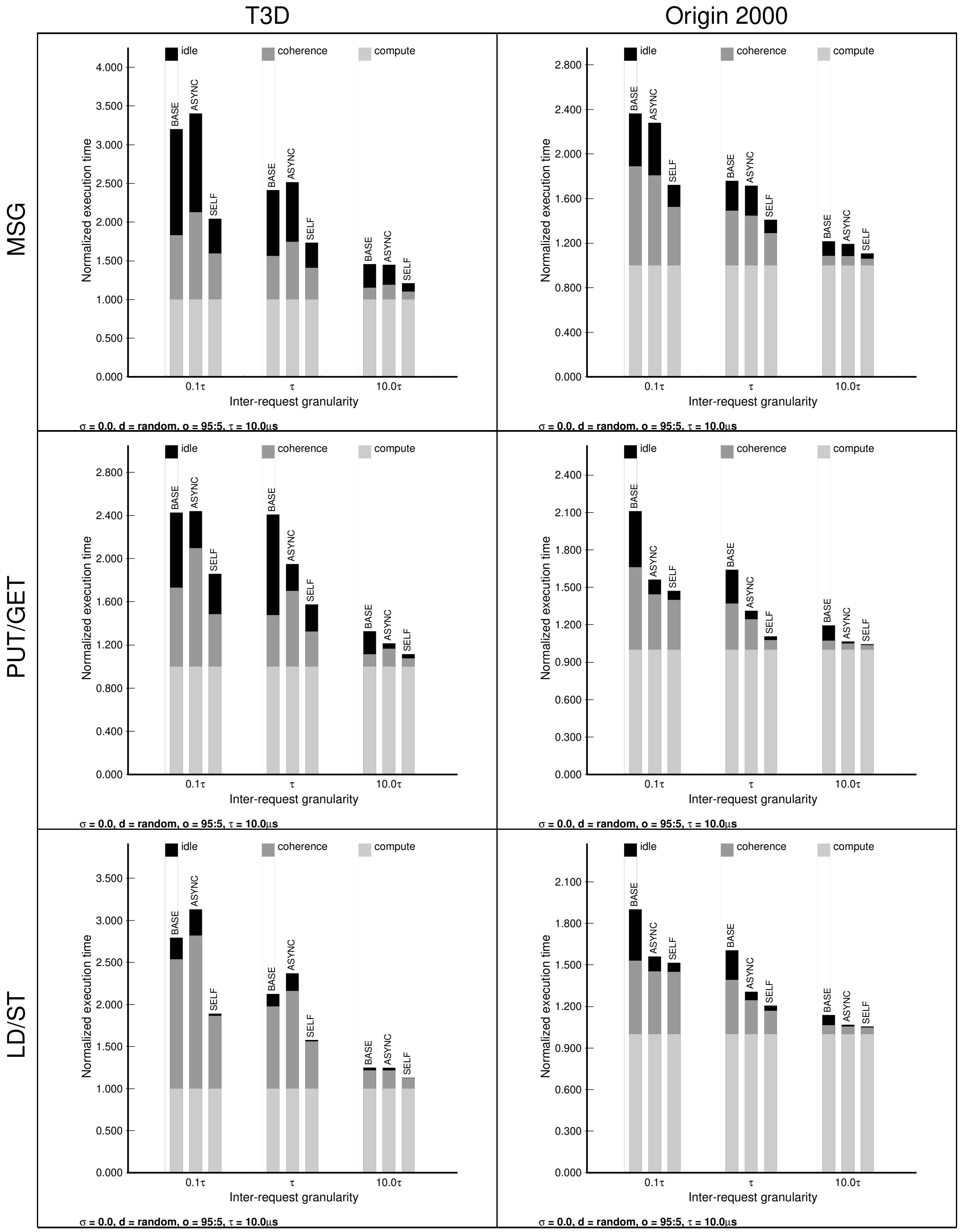

Figure 5.10 Sensitivity to responsiveness: Microkernel execution time breakdown on the T3D and the Origin showing impact of view caching protocols as interrequest compute granularity is varied.

both idle time and coherence overhead. These results suggest that a less sophisticated communication architecture can still yield good performance provided sufficient knowledge of application semantics is available.

PUT/GET. Unlike the MSG interface, both ASYNC and SELF protocols exhibit significant improvement in execution overheads over the BASE protocol. The improvement in both cases stems from using requester-initiated data transfer and reducing the involvement of the home node. The improvement on the T3D is limited by the increased per-operation overheads, due to the architectural disadvantage faced by remote loads as opposed to stores. For reasons discussed earlier, the Origin biases the advantage in the opposite direction which explains its increased magnitude of improvement. Thus, in contrast to the base protocols, the view caching protocols expose operations that are better able to exploit the low overhead responsive communication hardware to yield a marked performance advantage.

LD/ST. The ASYNC protocols are essentially identical to the BASE protocols, differing only in the amount of loss contention. These differences show up as minor performance differences. In contrast, the performance of the SELF implementation shows significant improvements, exhibiting the cumulative advantages of reduced coherence volume, reduced per-operation overheads, and reduced coupling between participating processors. The magnitude of improvements is limited by the fact that the BASE protocols also achieve good performance, being able to take advantage of the richer communication architecture features.

5.6.2 Sensitivity to occupancy

Figure 5.11 shows microkernel execution times for implementations of the two view coherence protocols on various communication architectures as the read-to-write ratio is varied, keeping other parameters constant as specified in Section 5.4.1. Figure 5.12 shows the corresponding graphs as the shared access distribution is varied. The graphs are organized as in the previous section.

The execution overhead improvements due to view coherence protocols exhibit similar trends as with the earlier microkernel configurations. The magnitude of performance improvement from custom protocols increases across all communication architectures as either the coherence traffic increases (when the read to write ratio is decreased from 95:5 to 85:15), or when the access distribution is skewed (from random to hotspot,$\times$8). Improvements in both cases arise primarily from the decoupling introduced by view coherence protocols, this decoupling reduces the aggregate occupancy by protocol handlers and hence performance degradation due to queuing effects. For example, on the PUT/GET

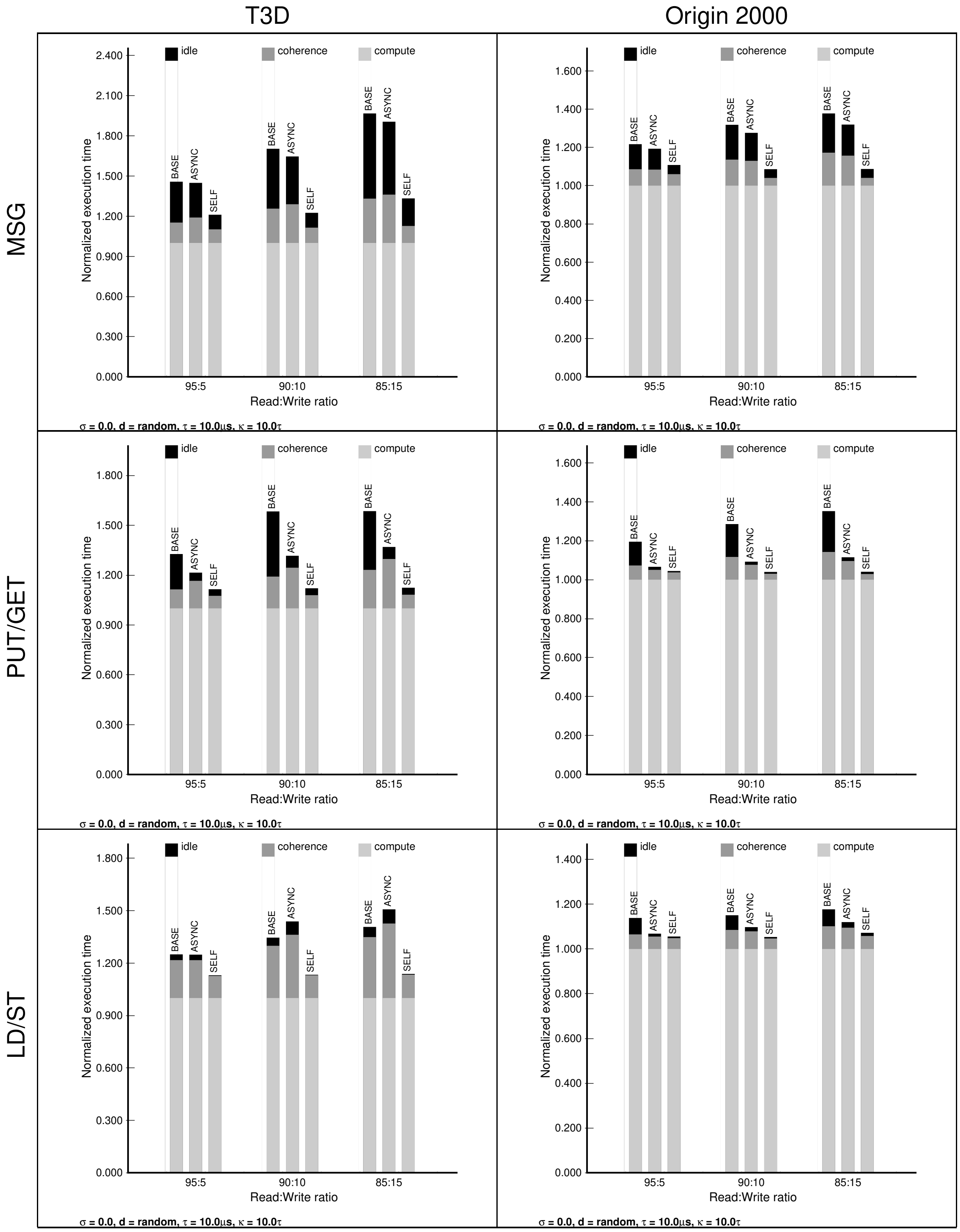

Figure 5.11 Sensitivity to occupancy: Microkernel execution time breakdown on the T3D and the Origin showing impact of view caching protocols as read-to-write ratio is varied.

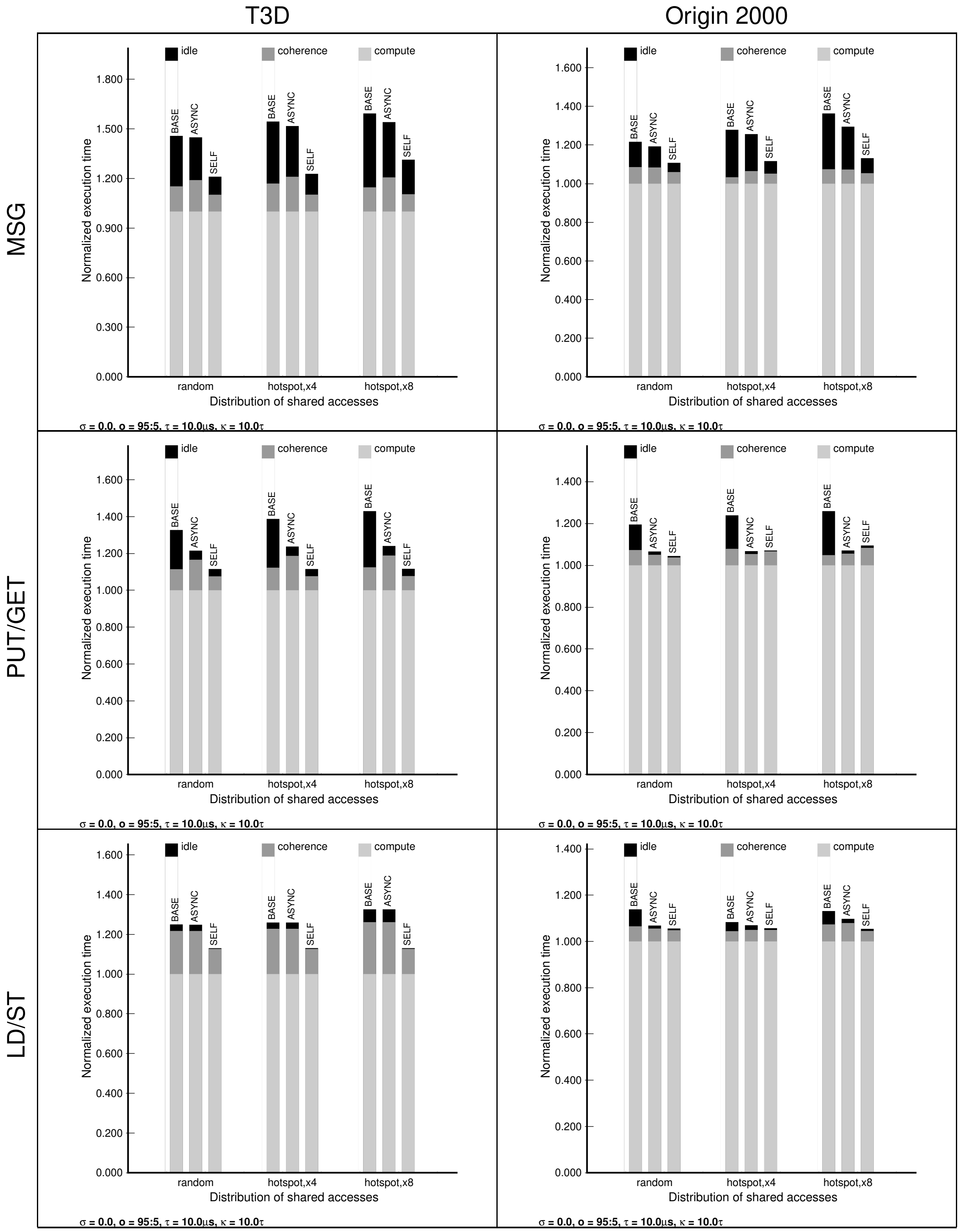

Figure 5.12 Sensitivity to occupancy: Microkernel execution time breakdown on the T3D and the Origin showing impact of view caching protocols as distribution of shared accesses is varied.

interface, unlike BASE whose overheads increase as the access distribution is skewed from random to hotspot$\times 8$, both ASYNC and SELF deliver performance which is virtually insensitive to the sharing access pattern. The main reason for this is the significantly reduced occupancy of ASYNC and SELF protocols.

5.6.3 Summary

In this section, we have shown that exploiting application knowledge semantics to build view coherence protocols that decouple processor activities and require reduced synchronization among them can improve microkernel performance across a range of communication architectures. Additionally, unlike existing DSM protocols, view caching protocols show reduced sensitivity to processor unresponsiveness and unbalanced sharing, even on simpler communication architectures such as MSG and PUT/GET. The improvements from view caching protocols arise more due to the qualitative advantages of the protocols (less synchronization and processor coupling) as compared to any per-operation quantitative benefits. In fact, for most organizations on the T3D, view caching protocols incur higher per-operation costs. Yet, they deliver better aggregate performance.

In addition to improvements in the context of individual communication architectures, it is interesting to observe the performance impact of view caching protocols across the range of architectures. Figure 5.13 summarizes the performance of base and view caching protocols on various communication architectures for one microkernel configuration. Two observations can be made from the graphs in the figure. First, the view-coherence protocols are better able to exploit a given level of hardware support (e.g., the PUT/GET organization). Second, view caching protocols on a particular communication architecture achieve performance parity with base protocols on a more sophisticated architecture, suggesting that the customizing protocols using knowledge of application semantics can compensate for a lack of hardware support.

5.7 Related Work

Our research has focused on developing data locality techniques which yield robust high performance for dynamic multithreaded computations. Our approach exploits knowledge of application data access semantics and leverages low overhead responsive operations in advanced communication architectures to construct latency-tolerant custom coherence protocols which require less processor coupling

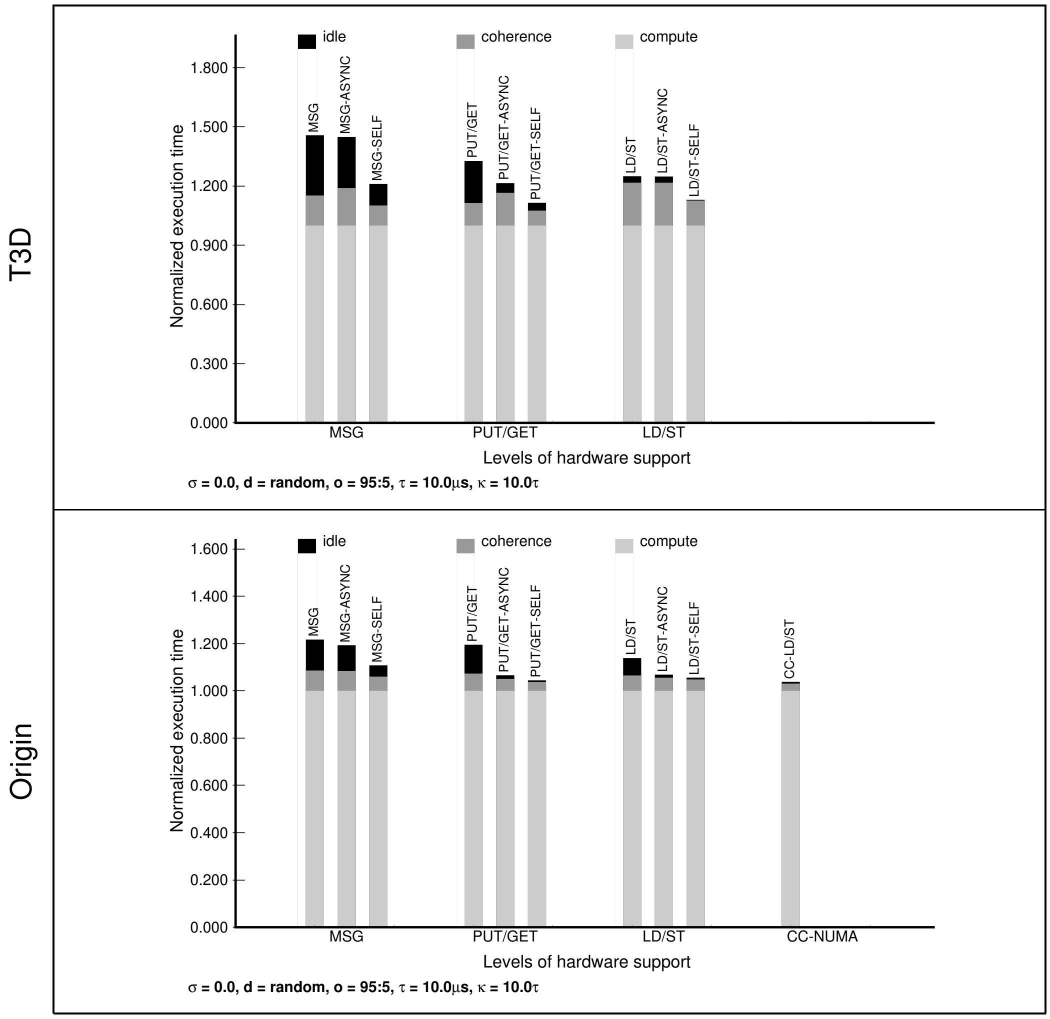

Figure 5.13 Microkernel execution time breakdown on the T3D and the Origin showing the impact of view caching protocols versus different communication architectures for a fixed set of parameters.

and are consequently more tolerant of unresponsive processors and unbalanced communication. Related work on software DSM systems can be classified into three categories: weak consistency models, hybrid hardware-software approaches, and application-specific protocol customization.

Weak consistency models. Several page-based [11, 101, 102] and object-based [77–79, 106–108] DSM systems have been proposed which rely on weak consistency models for performance. Such systems use request-reply protocols derived from hardware cache coherence systems, requiring prompt servicing of coherence requests (similar to dedicated hardware cache controllers). However, as our evaluation of object-consistent DSM in Section 5.4 demonstrates, the use of synchronous request-reply protocols shows high sensitivity to processor unresponsiveness and protocol handler occupancy. This sensitivity results in performance degradation on communication architectures such as MSG and PUT/GET where it increases the time processors idle awaiting completion of remote coherence actions.

A class of DSM systems [148, 149, 152] predicts access requests and eagerly creates an object copy on the requesting processor. These solutions eliminate coherence operations from the critical path and replace synchronous protocols with asynchronous ones. However, such optimizations are less effective in a dynamic context where predictive information about future object accesses is hard to obtain.

Hybrid hardware-software approaches. Recent work by several researchers [102, 112, 113] has examined the use of hardware support for improving the performance of page-based software DSM systems. Specifically, these researchers have explored how coherence protocols (and often new consistency models) can be developed to take advantage of hardware features such as remote stores and reflective memory. Although our approach shares some of the above goals, it differs in two important ways. First, instead of assuming a rich set of hardware primitives, we have identified the incremental qualitative and quantitative contribution from different levels of general-purpose communication architecture support. Second, we have chosen to work within the constraints imposed by a general fine-grained software DSM consistency model (object consistency) and examined what it takes to get good performance for a range of application scenarios.

Application-specific protocol customization. Systems such as Munin [28], Poly-C [29], and Tempest [30], which allow application-specific customization of coherence protocols, are most similar to our approach. Munin and Poly-C allows custom coherence actions for shared regions by selecting from among a set of predefined coherence protocols that reflect common application sharing scenarios. A related approach is also employed in the Charm system [95], which provides a predefined set of information-sharing mechanisms. The Tempest approach is more general, permitting an arbitrary user-defined coherence handler to be invoked to service an access request.

View caching is very similar to the above systems, yet there are important differences. View caching permits customization at the granularity of use (i.e., the view) as compared to subjecting all accesses to an object to the same coherence policy. As our application results in Chapter 7 show, treating each view as an independent coherence unit provides additional opportunities for optimization. In addition, dispatching to a general user-level handler as in Tempest is likely to incur relatively large overheads in the context of fine-grained dynamic computations. Unlike Tempest, which only provides the mechanisms for customization, our approach makes the case for a specific kind of customization—reducing synchronization requirements of coherence protocols—as being important for high performance DSM in the context of dynamic computations.

5.8 Summary

In this chapter we have described view caching, a latency-tolerant caching framework for improving data locality of dynamic multithreaded computations.

Our approach is motivated by the poor performance of existing software DSM implementations for dynamic applications. Experiments with a synthetic sharing microkernel show that a state of the art object-consistent DSM system does not provide performance robust over unresponsive processors and unbalanced sharing on either a MSG or a PUT/GET communication architecture interface. Since the latter includes low-overhead responsive communication operations, our results indicate that such operations on their own do not guarantee high performance; software protocols are also required which can exploit them for a performance advantage. Our microkernel results suggest that the reduced sensitivity of the LD/ST interface to responsiveness and occupancy may be crucial for obtaining good performance for dynamic computations using existing DSM protocols. We corroborate this conclusion using application programs in Chapter 7.

The view caching approach exploits knowledge of application data access semantics and leverages low overhead responsive operations in advanced communication architectures to construct latency-tolerant custom coherence protocols which deliver performance robust over unresponsive processors and unbalanced communication. The view caching framework decomposes coherence operations into three components—`access-grant`, `access-revoke`, and `data-transfer`—allowing custom protocols to be built by composing specialized implementations of each component which require less synchronization and processor coupling, as well as can effectively exploit varying levels of communication architecture hardware support. Knowledge of application semantics drives the choice of the most efficient component implementation. Microkernel results show that view-caching protocols demonstrate reduced sensitivity to processor unresponsiveness and unbalanced sharing, even on less sophisticated communication architectures such as MSG and PUT/GET, despite higher per-operation costs. In addition, view caching protocols on a particular communication architecture achieve performance parity with base protocols on a more sophisticated architecture, suggesting that the customizing protocols using knowledge of application semantics can compensate for a lack of hardware support.

CHAPTER 6

LOAD BALANCE

In this chapter, we describe a framework for load balancing dynamic multithreaded computations. Our framework combines run-time object placement and thread placement heuristics to efficiently distribute threads with wide variations in granularity, while preserving data locality and application execution priorities.

This chapter is organized as follows. Section 6.1 discusses the load-balancing problem in the specific context of our computation model and provides an overview of our solution. In Section 6.2, we describe a simple model of dynamic task[1] generation in a global object space and prove the intractability of two formulations of the load-balancing problem. These results motivate the heuristic framework introduced in Section 6.3. Our approach organizes object and thread placement heuristics in a cost hierarchy; a compiler or a programmer selects the most efficient policy, which matches the task-generation attributes of an application. Three components of the framework are described in Sections 6.4 – 6.6: Section 6.4 describes required language and run-time support, Section 6.5 presents a library of object and thread placement policies, and Section 6.6 describes a microkernel-based evaluation of the suitability of these policies for specific application archetypes. Finally, Section 6.8 summarizes our contributions.

6.1 Background

The goal of load balancing is to effectively utilize the available processors with the concurrency available in a parallel application. Our focus here is on ensuring that no processor is idle as long as there is concurrent work in the system. Although, the simplest approach to load balance is to statically distribute tasks, such an approach will not be successful with dynamic multithreaded computations

[1] The terms *task* and *thread* are used interchangeably in this chapter.

because computation granularity and dependences cannot be accurately predicted by static analyses. Consequently, we restrict our attention to dynamic load-balancing techniques, which determine the mapping of tasks to processors at run time.

In this section, we first describe the problem constraints imposed by the computation model, and then provide an overview of our solution.

6.1.1 The problem

The load balancing problem for dynamic multithreaded computations is distinguished by the following characteristics of the computation model and irregular applications (see Section 3.1):

- *Wide variations in thread granularity:* Thread granularity variations imply that load balancing mechanisms must have low overhead to support and efficiently distribute fine-granularity threads. Additionally, these mechanisms must be robust over computation irregularity created by the granularity variations.

- *Locality sensitivity:* Threads require access to shared data objects. Therefore, load balancing mechanisms must preserve data locality to achieve execution efficiency.

- *Priority sensitivity:* Some irregular search applications require prioritization of thread execution, so load balancing mechanisms must preserve these priorities to avoid creating redundant work.

Previous solutions [6, 7, 9, 10, 13, 24, 27, 117–120] described in Section 3.3 are valid for systems which exhibit only a subset of these characteristics. Most are either too heavyweight for use with fine-granularity threads [6, 10, 13, 117–120], completely ignore locality [7, 24], or work well only with regular computations [9, 10, 27]. Our solution provides a general framework that enables load balance for systems with all of the above characteristics.

6.1.2 Solution overview

The described characteristics imply that mechanisms for load balancing must simultaneously incur low overhead, and be sophisticated enough to ensure that thread distribution respects data locality and thread execution priority. Our solution satisfies these conflicting objectives by providing a load-balancing framework which contains a collection of mechanisms of differing costs and capabilities.

Instead of dictating the use of one mechanism for all threads in the computation, our approach relies on a compiler or a programmer to select the most efficient mechanism suitable for different sets of threads.

The load balancing framework consists of three components:

1. Language and run-time system support to (a) associate a subset of threads with a specific load balancing mechanism, and (b) efficiently synthesize application load balance from multiple load balancing policies within the same application.

2. A library of object placement, thread placement, and thread-ordering policies.

3. A correspondence between application attributes and load balancing policies. Identifying which policy is suitable for a given application archetype provides guidance to the compiler or programmer.

The overall framework is presented in Section 6.3, with details of the individual components described in Sections 6.4, 6.5, and 6.6 respectively.

6.2 Intractability of the Load-Balancing Problem

In this section, we first describe the task structure in dynamic multithreaded computations to formally define the load-balancing problem. We then show the intractability of two formulations of the load-balancing problem. These intractability results motivate the use of heuristics in our approach.

6.2.1 Task structure in dynamic multithreaded computations

A formal definition of the load balancing problem requires an understanding of the task structure in dynamic multithreaded computations. Figure 6.1 shows pictorially the main elements of the computation which are described in detail below:

Processors $P_1 \ldots P_N$**.** The system contains a number of processors, each of which has direct access to a private memory. Access to another processor's memory requires a communication operation. In the figure, processors are represented using light gray rectangles.

Objects $D_1 \ldots D_m$**.** An object resides in a single processor's memory. An object can have several copies in the system: Each copy resides in a single processor's memory and is associated with a single data object. The number of object copies in a processor's memory may be limited by the size of a cache

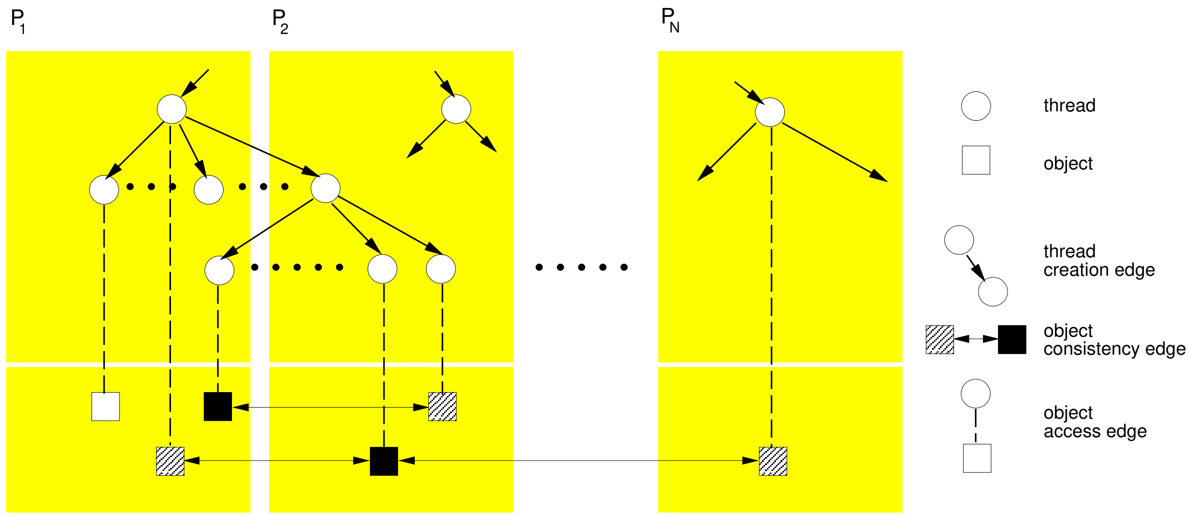

Figure 6.1 Task structure in dynamic multithreaded computations.

designated for holding copies of remote objects. Object copies are kept coherent using a consistency protocol of the kind described in Chapter 5. In the figure, object copies are represented using small rectangular boxes: A white box represents an object with no copies, a black box represents the home copy of an object, and the other boxes represent nonhome copies. The association between a copy and its home object is shown using object consistency edges.

Threads $T_1 \dots T_n$**.** A thread resides on a single processor and can access one or more data objects. Threads are shown using circles in the figure, and associations between threads and objects are shown using the object access edges. Threads can dynamically create other threads, shown in the figure using thread creation edges. Thread execution on a processor requires valid local copies of objects it accesses. Obtaining a valid object copy may involve a communication operation, and invalidation of other copies in the system.

During its lifetime, a thread goes through various states. A thread starts off in an *embryonic* state on creation and then moves into a *waiting* state, where it waits for access to object copies. The embryonic state models the communication time either for the thread to be shipped to a processor where the data object or one of its copies exist (*function-shipping*), or to obtain copies of all data objects needed for thread execution on the current processor (*data-shipping*). The waiting state models the wait time due to concurrency control. Once the thread has been granted access, it becomes *ready*, where it waits pending a scheduling action by the processor scheduler. The processor scheduler interleaves processor resources among ready threads. The scheduling step causes the selected thread to enter an *active* state, where it can operate against object state, create new objects, and spawn other threads. A thread *suspends* when it needs to access the return values of spawned threads. Later, on receiving all the return values, the thread reenters the *ready* state. The total execution time of the thread is determined by the time spent in each of these states. The program execution time equals the completion time of the last thread.

Given the above task structure, the load balancing problem can be stated as follows:

> Given an initial layout of data objects (including a policy for dynamically created objects), to construct an execution schedule for threads $T_1 \ldots T_n$ which minimizes the program execution time. An execution schedule specifies both the processor on which a thread executes, and the order of its execution.

6.2.2 Intractability results

We show that the general load-balancing (LB) problem as stated above is intractable by proving that two strictly simpler static formulations of the problem are NP-complete [153]:

1. LB-PARALLEL: Given processors $P_1 \ldots P_N$ with infinite sized caches, data objects $D_1 \ldots D_m$, and independent tasks $T_1 \ldots T_n$ which require execution times $t_1 \ldots t_n$, respectively, and each task uses exactly one data object. There is no communication cost. The optimization problem is to schedule the tasks so as to minimize the completion time of the last task. The corresponding decision problem can be formulated as follows: Can a schedule be generated such that the completion time of the last task is less than some positive constant U?

2. LB-CACHE: Given a single processor P with a cache of size C, independent unit execution time tasks $T_1 \ldots T_n$, and data objects $D_1 \ldots D_m (C < m)$ such that each task uses exactly C objects. There is unit cost for moving an object into and out of the cache, and task execution requires that all its accessed objects be resident in the cache. The optimization problem is to schedule the tasks so as to minimize the completion time of the last task. The corresponding decision problem can be formulated as follows: Can a schedule be generated such that the completion time of the last task is less than some positive constant U?

These two formulations model the two main challenges of the general load-balancing problem: parallelism and locality. LB-PARALLEL models the parallelism challenge, and is a classical scheduling formulation. The locality aspect of the load-balancing problem is modeled by the LB-CACHE formulation which is less well known and has only been informally described in literature [154].

To show that the general load balancing problem is intractable, we prove that the above formulations are NP-complete. Given the ubiquity of the LB-PARALLEL formulation, we only sketch the proof of its NP-completeness. Since we were not able to find a formal proof of the intractability of LB-CACHE, we describe its proof in greater detail.

Theorem 6.1 *LB-PARALLEL is NP-complete.*

Proof: The NP-completeness of the decision problem follows by reduction from the SET-PARTITION problem [153]: An instance of the above decision problem can be constructed from an instance of the SET-PARTITION problem by considering the numbers to be task times and setting U to be $\sum t_i/2$. Note that this reduction does not rely on the data objects at all. Thus, the general load-balancing problem is intractable, even if tasks do not access any data objects, or if all accessed objects are available on all processors. ■

Theorem 6.2 *LB-CACHE is NP-complete.*

Proof: We prove the NP-completeness of the LB-CACHE formulation by first mapping it into a variant of the traveling salesperson problem (TSP) on Hamming spaces, HAMMING-TSP, and then proving the intractability of HAMMING-TSP by transforming a known NP-complete problem, TSP, into it. The TSP and HAMMING-TSP problems can be formally stated as below:

Hamming Space: F_2^N is the space of binary vectors of length N.

Hamming Distance: $hd(x_i, x_j)$ is the number of 1's that are different in the binary representations of x_i and x_j.

TSP: Given positive integers D, R, a set of vertices $V = \{v_1, v_2, \ldots, v_n\}$, and integer distances between pairs of vertices $d(v_i, v_j)$ such that $\max_{i,j} d(v_i, v_j) \leq D$.

Is there a tour that includes all the vertices $V_\pi = \langle v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)} \rangle$ such that the tour length $\sum_{i=1}^{n} d(v_{\pi(i)}, v_{\pi(i+1)}) \leq R$, where π is a permutation of $\{1, \ldots, n\}$, and $\pi(n+1)$ is understood as $\pi(1)$ in the above sum?

This formulation of the TSP problem is strongly NP-complete.

HAMMING-TSP: Given positive integers $N, M \leq 2^N, C \leq N, K$, and a set of vectors $X = \{x_1, x_2, \ldots, x_M\}$ where for all i, $x_i \in F_2^N$ and the Hamming weight (the number of 1's in the vector) of every x_i is C.

Is there an ordering of vectors $X_\sigma = \langle x_{\sigma(1)}, x_{\sigma(2)}, \ldots, x_{\sigma(M)} \rangle$ such that

$$S_\sigma(X) = \sum_{i=1}^{M} hd(x_{\sigma(i)}, x_{\sigma(i+1)}) \leq K$$

where σ is a permutation of $\{1, \ldots, M\}$, and $\sigma(M+1)$ is understood as $\sigma(1)$ in the above sum?

The mapping between LB-CACHE and HAMMING-TSP uses the fact that each task in the LB-CACHE problem corresponds to a binary vectors of length m, with 1's in the positions corresponding to data objects that are accessed by the task. The problem of finding a schedule for the tasks that minimizes completion time is equivalent to finding a *minimum cost path* traversing the corresponding vectors in the Hamming space. The cost of going from one task to the next, the Hamming distance, is the cost of ejecting unused data objects from the cache and bringing in the required new objects. Note that HAMMING-TSP corresponds to finding the *minimum cost tour* instead of the minimum cost path, but the path and tour problems are interreducible. Formally, the integers N, M, C, K of the HAMMING-TSP instance correspond to the integers m, n, C, U of the LB-CACHE instance with the binary vectors constructed as described above.

It is easy to see that HAMMING-TSP $\in$ NP, since a nondeterministic algorithm need only guess an ordering of vectors σ and check in polynomial time whether $S_\sigma(X) \leq K$.

To show that HAMMING-TSP is NP-hard, we transform TSP to HAMMING-TSP. The transformation uses the following intuition (see Figure 6.2): The n vertices of the TSP instance are replaced by n components (each component is a collection of $(n-1)$ vectors) in the HAMMING-TSP instance, such that a tour in the TSP instance corresponds to an ordering of the vectors in the HAMMING-TSP instance. The transformation ensures that an ordering of vectors that satisfies the bound K has all the vectors of a component in contiguous positions (i.e., the entire component is picked up as a unit), and the order of traversal of components corresponds to the traversal of vertices in the TSP instance.

We need to specify an instance of HAMMING-TSP, in particular the positive integers $N, M \leq 2^N, C \leq N, K$ and a set X of M vectors in the binary hamming space F_2^N. We define

$$N = np + \frac{n(n-1)}{2} r$$

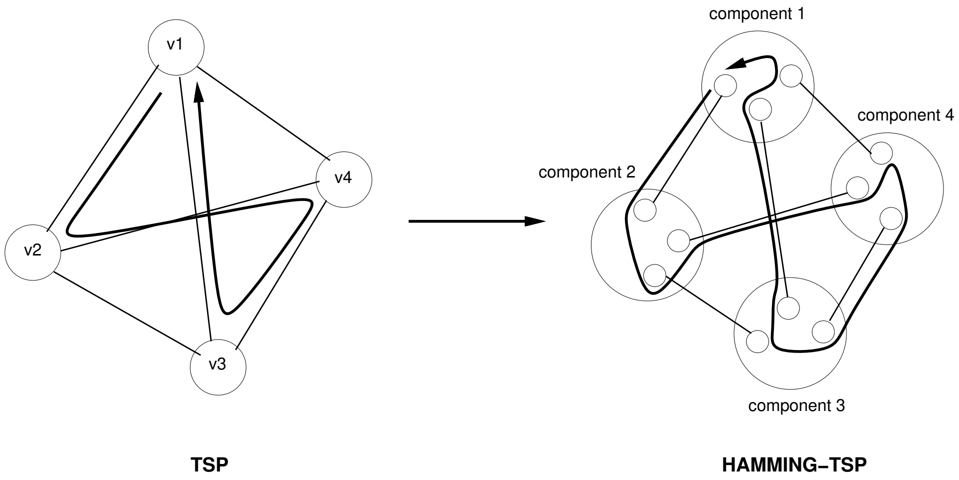

Figure 6.2 Intuition for transforming TSP into HAMMING-TSP.

$$\begin{aligned} M &= n(n-1) \\ C &= p+q \\ K &= 2pn + 2qn(n-2) + 2B \end{aligned}$$

where p, q, r are integer constants whose values will be determined later. The M vectors are organized into n components of $(n-1)$ vectors each. The vectors in component i are labeled $x_{i,j}$, for $1 \leq j \leq n, i \neq j$. For example, component 1 consists of the set $\{x_{1,2}, x_{1,3}, \ldots, x_{1,n}\}$, and component 2 of the set $\{x_{2,1}, x_{2,3}, \ldots, x_{2,n}\}$.

The key to the transformation is a bit representation for each of the vectors, shown pictorially in Figure 6.3. Each $x_{i,j}$ is a binary vector of length N, organized into n zones of p bits apiece ($A_1, A_2, \ldots, A_n$), and $n(n-1)/2$ zones of r bits apiece ($B_{1,2}, B_{1,3}, \ldots, B_{n-1,n}$). Vector $x_{i,j}$ has 1's for all the p bits corresponding to zone A_i, and q of the r bits of zone $B_{\min(i,j),\max(i,j)}$; all the remaining bits are set to 0. The q bits are chosen so that the $B_{i,j}$ zone contributes $2d(v_i, v_j)$ to the Hamming distance between the two vectors $x_{i,j}$ (in the component corresponding to TSP vertex v_i) and $x_{j,i}$ (in the component corresponding to TSP vertex v_j). This can be achieved by taking any q of the 1's in zone $B_{i,j}$ for vector $x_{i,j}$, and generating vector $x_{j,i}$ by replacing $d(v_i, v_j)$ of the 1's with 0's and the same number of 0's with 1's. r needs to be large enough to allow this:

$$r \;\geq\; (q + \max_{i,j} d(v_i, v_j)) \;\geq\; (q + D) \tag{6.1}$$

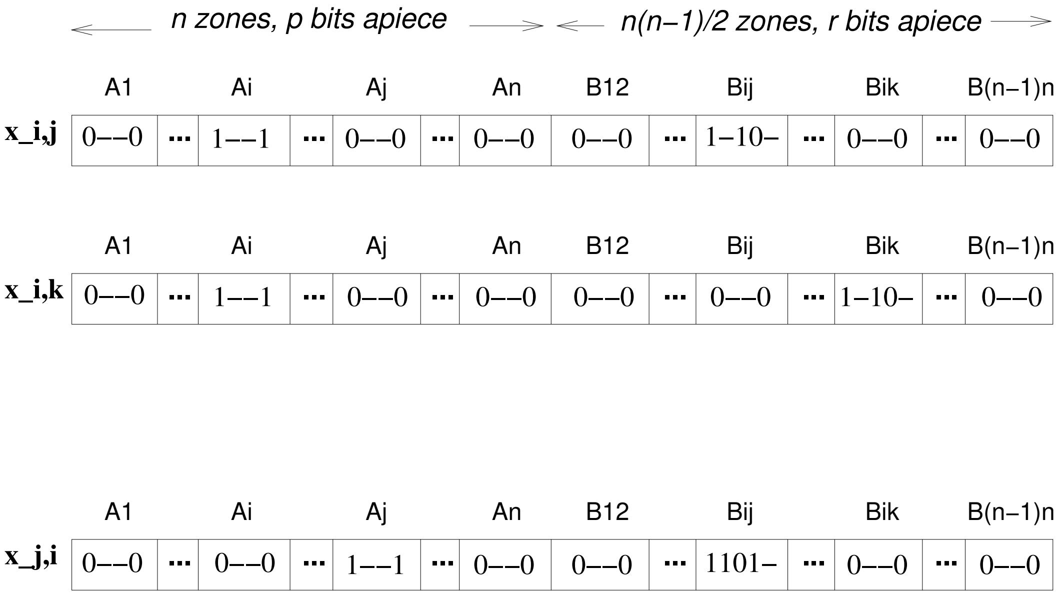

Figure 6.3 Bit representation of HAMMING-TSP vectors.

The above representation has the following properties:

1. The Hamming weight of each vector is $p + q$.

2. Two vectors in the same component ($x_{i,j}$ and $x_{i,k}, j \neq k$) are at Hamming distance $2q$. This follows because the two vectors have the same bits set among the A zones (in A_i), and have q bits set in different B zones (in $B_{\min(i,j),\max(i,j)}$ and $B_{\min i,k,\max i,k}$).

3. Two vectors in different components that share the same B zone, $x_{i,j}$ and $x_{j,i}, j \neq i$ are at Hamming distance $(2p + 2d(v_i, v_j))$. This is ensured by the fact that the two vectors have bits set in different A zones (which contributes $2p$ to the distance), and the q bits that are set to 1 are chosen to satisfy the remaining portion of the distance $(2d(v_i, v_j))$.

4. Two vectors in different components that do not share the same B zone are at Hamming distance $(2p + 2q)$. This follows because the vectors do not share either the A or the B zones.

The values of p and q are chosen so that for any ordering X_σ of vectors in the HAMMING-TSP instance, $S_\sigma(X) \leq K$ if and only if X_σ satisfies the following properties:

1. all vectors in a component appear in contiguous positions in X_σ.

2. if vectors of component i immediately precede vectors of component j in X_σ, then vector $x_{i,j}$ immediately precedes vector $x_{j,i}$.

To ensure the first property of X_σ, we select p such that it penalizes splitting up a component in the ordering. Each time a component is entered or exited, the contribution to $S_\sigma(X)$ is at least $2p$ (from properties 3 and 4 of the representation). Since each component must be entered and exited at least once in a tour, $S_\sigma(X) \geq 2pn$. If a component is split up, i.e., its vectors do not appear contiguously in X_σ, a component appears more than once in the tour: The additional contribution from each appearance (one entry and one exit) is at least $4p$. Thus, choosing p such that

$$4p \quad \geq \quad (2qn(n-2) + 2B) \tag{6.2}$$

ensures that no component is split up in X_σ.

To ensure the second property of X_σ, we choose q such that it penalizes transitions between components i and j in X_σ using any pair of vectors other than $x_{i,j}$ and $x_{j,i}$. The Hamming distance between $x_{i,j}$ and $x_{j,i}$ is $(2p + 2d(v_i, v_j))$ (property 3 of the representation), and that between all other pairs of vectors is $(2p + 2q)$ (property 4 of the representation). Thus, choosing q such that

$$2q \quad \geq \quad 2d(v_i, v_j), \text{for all } i \neq j \tag{6.3}$$

ensures that $S_\sigma(X) \geq K$ if the transition between components happens with any pair of vectors other than $x_{i,j}$ and $x_{j,i}$. Since $S_\sigma(X)$ is assumed to be at most K, transitions between components i and j must involve the $x_{i,j}$, $x_{j,i}$ pair.

Choosing

$$\begin{aligned} r &= 2D + 1 \\ p &= \frac{(D+1)n^2 + B}{2} \\ q &= D + 1 \end{aligned}$$

satisfies the constraints in Equations (6.1), (6.2), and (6.3), respectively.

This transformation can clearly be completed in polynomial time, since both the number of $x_{i,j}$, as well as the length of each $x_{i,j}$ are bounded by a polynomial involving n, D, and B.

To complete the proof, we need to prove that the above is indeed a transformation, i.e., there exists a tour of length $\leq R$ in the TSP instance if and only if there exists an ordering of vectors in the HAMMING-TSP instance such that $S_\sigma(X) \leq K$:

Proof of $\Rightarrow$: If there exists an ordering of vectors X_σ in the HAMMING-TSP instance such that $S_\sigma(X) \leq K$, by construction such an ordering must traverse all vectors in a component contiguously with the transition between components i and j using vectors $x_{i,j}$ and $x_{j,i}$. Thus, the ordering on vectors X_σ defines an ordering on components π that corresponds to a tour in the TSP instance. To verify that the tour corresponding to π satisfies the length bound R, we see that

$$S_\sigma(X) = (\underbrace{2q(n-2)}_{\text{intra-component}})n + \sum_{i=1}^{n}(\underbrace{2p + 2d(v_i, v_{\pi(i)})}_{\text{inter-component}})$$

Since $S_\sigma(X) \leq K$, substituting for K and eliminating constant terms on both sides of the inequality, we obtain

$$\sum_{i=1}^{n} d(v_i, v_{\pi(i)}) \leq R$$

which proves our result. ■

Proof of $\Leftarrow$: If there exists a tour π in the TSP instance with length $\sum_{i=1}^{n} d(v_i, v_{\pi(i)}) \leq R$, we can construct an ordering of vectors X_σ in the HAMMING-TSP instance which (i) traverses the components in the same order as the vertices in the TSP instance, (ii) goes between components i and $j = \pi(i)$ using vectors $x_{i,j}$ and $x_{j,i}$, and (iii) includes the other vectors within the same component in any order. Corresponding to this ordering,

$$\begin{aligned} S_\sigma(X) &= (\underbrace{2q(n-2)}_{\text{intra-component}})n + \sum_{i=1}^{n}(\underbrace{2p + 2d(v_i, v_{\pi(i)})}_{\text{inter-component}}) \\ &= 2pn + 2qn(n-2) + 2\sum_{i=1}^{n} d(v_i, v_{\pi(i)}) \\ &\leq 2pn + 2qn(n-2) + 2R \\ &\leq K \end{aligned}$$

which proves our result. ■

Thus, the HAMMING-TSP and hence, LB-CACHE problems are NP-complete. This result implies that the general load-balancing problem is intractable even if we restrict ourselves to executing all tasks on a single processor. ■

Summary. In this section, we have proved the intractability of two simple static formulations of the load balancing problem. The intractability of the general load balancing problem follows from these results because the general problem exhibits significant additional complexity along several dimensions:

1. The general problem requires both the parallelism and locality aspects of the problem to be addressed simultaneously.

2. Unlike the unit cost model for task granularities and object access costs, the general problem exhibits wide variations in thread and object costs, that are execution dependent.

3. The general problem is characterized by dynamic creation of objects and threads, so its solution would require an online algorithm.

4. Object caching is subject to sharing based invalidations, adding significant complexity to the real problem when compared with the caching model assumed in LB-CACHE.

5. Finally, threads are subject to object-level concurrency control, which introduces additional constraints on thread scheduling.

The intractability of the general load-balancing problem motivates use of heuristics in the framework described next.

6.3 Load-Balancing Framework

In this section, we describe a heuristic framework for load-balancing dynamic multithreaded computations. Given the conflicting goals of low-overhead mechanisms, and priority and locality sensitivity, a single load balancing mechanism is not sufficient for the entire computation. Instead, our framework provides a collection of object placement, thread placement, and thread ordering policies of differing costs and capabilities. A compiler or a programmer selects the most efficient policy suitable for each set of threads.

We first present the overall strategy for load balancing, and then describe the framework components required to implement the strategy.

6.3.1 Strategy for load balancing

Our strategy for load balancing an irregular computation views the overall problem as several sub-problems, each of which achieves load balance for a set of the threads. A natural way of referring to a set of threads is to group all threads created at the same static *thread creation site*, i.e., the method invocation site in the source program. Our strategy decides for each thread creation site in the program, how the threads created there will be load balanced, and then synthesizes application load balance by combining these multiple load balancing mechanisms.

Figure 6.4 depicts this strategy. The threads in the figure are grouped into three sets and are shown using white, cross-hatched, and black circles. Each set of threads is balanced using a potentially different load-balancing policy. For example, the threads shown in white circles are uniformly distributed using object placement, while the others rely on explicit thread-placement mechanisms which are managed using separate structures. The underlying implementation composes the three load balancing policies to achieve overall application load balance.

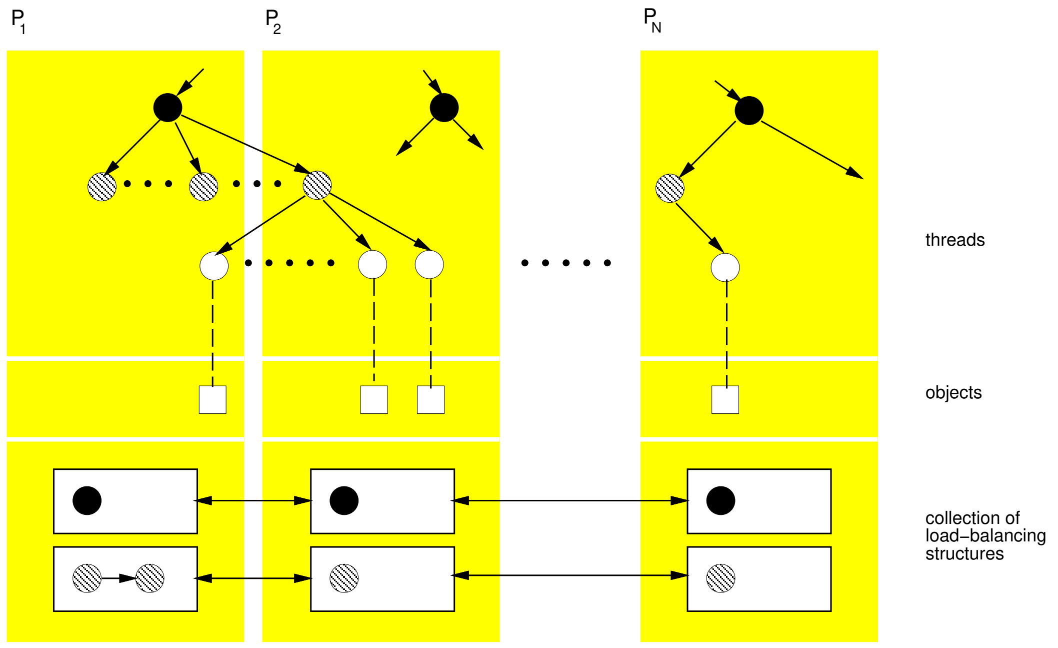

Figure 6.4 Overall load balancing strategy: Different subsets of threads (shown by the white, cross-hatched, and black circles) are subject to different load-balancing policies.

6.3.2 Components of the framework

The load-balancing framework consists of the above strategy and the components required to implement it. There are three implementation components:

1. Language and run-time system support to (a) associate a set of threads with a specific load-balancing mechanism, and to (b) synthesize application load balance by composing multiple load-balancing mechanisms within the same application.

2. A library of object placement, thread placement, and thread-ordering policies.

3. A correspondence between application attributes and load-balancing policies.

Language support permits the programmer to associate a set of threads to a specific load-balancing policy. This association is essential because all threads in the computation are not implicitly subject to a single load-balancing mechanism; consequently, the programmer must distinguish policies for different sets of threads. In practice, the programmer needs to specify only variations from a default policy. The run-time support synthesizes application load balance from the multiple policies, allowing a programmer or compiler to separately restrict attention to load balancing a subset of threads at a time.

The library of load-balancing policies provides separate mechanisms to allow different threads in the computation to be controlled by different policies, based upon considerations of thread granularity, irregularity, and required sensitivity to locality and priority. Object-placement mechanisms distribute threads by controlling the location of data objects and having threads execute close to data that they access. These mechanisms represent a low-cost solution to the load-balancing problem because thread distribution incurs low decision and scheduling overheads. Thread-placement mechanisms enable thread distribution independent of object placement, and can additionally be sensitive to data locality, albeit with increased cost. Thread-ordering mechanisms incur higher cost than thread placement, providing load balance while prioritizing thread execution.

The third element of the framework identifies the suitability of different load balancing policies given a specific set of application attributes. This correspondence provides guidance to a compiler or programmer for associating sets of threads with different load balancing policies.

These three components are described in Sections 6.4, 6.5, and 6.6, respectively.

6.4 Language and Run-Time Support

In this section, we describe the language and run-time support required by the framework. We first discuss ICC++ language support for associating a thread creation site with a specific load balancing policy, and then support in the Concert run-time system for composing these individual policies.

6.4.1 Language support: Associating load balancing policies

Language support identifies the set of threads that will be subject to a particular load-balancing policy and creates the association. We discuss in turn, support for associating threads with object placement, thread placement, and thread-ordering mechanisms.

Object placement. These mechanisms indirectly balance the threads created at a program site by first controlling the placement of a distinguished data object accessed by these threads, and then ensuring that the thread executes local to this object. Language support in ICC++ uses standard C++ placement syntax with the `new` operator to control the placement of individual objects and the distribution of object collections. An implementation-specific function `location_of` is assumed to be available which provides the run-time location of its argument. Various kinds of placement control are illustrated in the code fragment below:

```
Foo *obj1 = new ( LOCAL ) Foo(4);                // object placement
Foo *obj2 = new ( location_of(obj1) ) Foo(5);    // object collocation
Accum *coll1 = new ( BLOCK ) Accum[16];          // collection distribution
```

Note that the above support can also be used to control data locality. Here, our interest is in using object placement as a "wire frame" that controls the processor where a thread executes.

Thread placement. These mechanisms directly control the load balancing of the thread subset. Language support consists of annotations for thread creation sites. The annotations use the `!scheduler` keyword which takes as an argument one of the policies from the collection described in Section 6.5. The annotation specifies that all threads created at the site will be subject to the indicated policy. For example, in the following code fragment, threads created at the creation site are load balanced using a `WorkSharing` scheduler:

```
...
( thread_function( arg1, arg2, ... ) !scheduler WorkSharing );
...
```

Thread ordering. These mechanisms directly control both the processor on which a thread executes (to ensure load balance and preserve locality), and its execution order (to preserve priority). Language support for ordering is similar to that for thread placement mechanisms: The scheduler is specified using the `!scheduler` keyword. In addition, the `!priority` keyword takes an integer argument that specifies the priority of a specific thread. In the following code fragment, the thread corresponding to `thread_function` executes with priority `arg1` and is managed by a `WorkSharingPriority` scheduler.

```
...
( thread_function( arg1, arg2, ... ) !priority arg1 !scheduler WorkSharingPriority );
...
```

6.4.2 Run-time support: Synthesizing application load balance

Run-time achieves overall application load balance by composing the individual load-balancing policies specified for each set of threads. The support ensures two things. First, that each thread is distributed according to the appropriate load-balancing policy. And second, that different policies coexist with each other to ensure overall load balance.

Run-time support for individual object placement, thread placement and thread-ordering policies are built on top of the default policies for object and thread creation in the Concert run-time system. The default object creation policy is to allocate an individual object on a random processor, and distribute collections in a cyclic fashion. Thread execution uses a first-in first-out (FIFO) scheduler, and by default threads are executed on the processor containing the target object of the method invocation.

Object-placement policies adhere to the default thread-placement policy, so their integration does not require any additional support in the run-time system.

On the other hand, explicit thread placement and thread-ordering policies do require additional support, provided by a *custom scheduler framework*. The custom scheduler framework augments the default FIFO scheduler of the Concert run-time system with a set of application-specific schedulers as shown in Figure 6.5. On creation, each thread is enqueued into the appropriate scheduling structure. Each nonempty custom scheduler is periodically assigned scheduling quanta by the FIFO scheduler. It uses these quanta to execute the set of enqueued threads according to a specific load-balancing policy. Coexistence between different schedulers is ensured by a contract of operation (see below) which prevents any individual scheduler from receiving an inequitable share of processing resources. Thus, the custom scheduler framework provides support similar to operating system schedulers in real-time and multimedia systems [155–160] which manage tasks with different processing requirements. The primary difference of our approach is that the integration between different schedulers is considerably more efficient, relying on nonpreemptive scheduling of cooperating application threads.

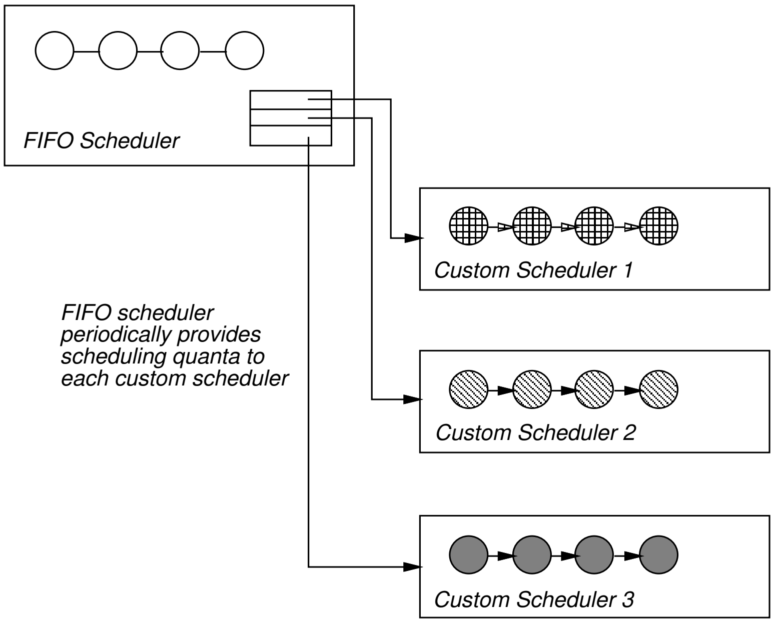

Figure 6.5 The custom scheduler framework augments the default FIFO scheduler with a set of application specific custom schedulers, each of which manages a set of threads according to a different load-balancing policy.

The Concert run-time system provides a predefined library of custom schedulers which implement the collection of load-balancing policies described in Section 6.5. In addition, programmers can write and integrate application-specific load balancing policies into the run-time system. To facilitate the latter, we describe three components of the custom scheduler framework: the interface for adding custom schedulers to build the hierarchy, the generic interface that each such scheduler must subscribe to, and the contract between the scheduler implementation and the run-time system.

Adding custom schedulers. Custom schedulers are added into the scheduling hierarchy using the following functions:

```
void init_custom_schedulers( Scheduler *parent );
void attach_scheduler( Scheduler *parent, CustomScheduler *sched );
```

The `init_custom_schedulers` function is a hook in the application program to allow specification of one or more custom schedulers. Each scheduler is added into the hierarchy using the `attach_scheduler` function. The order of calls to `attach_scheduler` specifies a linear priority ordering among the schedulers: The default FIFO queue has the highest priority.

Custom scheduler interface. Each custom scheduler implementation must provide the following functions, all of which are called as required by the run-time system:

```
void enqueue( CustomScheduler *sched, ... );
int  alive( CustomScheduler *sched )
int  size( CustomScheduler *sched );
void dispatch( CustomScheduler *sched, Scheduler *parent );
int  quiescence_callback( CustomScheduler *sched );
```

The `enqueue` function is called to enqueue a given thread according to the designated load balancing policy. The remaining functions serve as a handshake for implementing the contract discussed below. The `alive` function returns TRUE if the scheduler is active, FALSE otherwise. The `size` function returns the number of threads queued in the scheduler that are waiting for processing resources. A call to `dispatch` grants a scheduling quantum for the scheduler. Finally, the run-time system invokes the `quiescence_callback` function for a custom scheduler each time it detects quiescence.

Contract of operation. The contract between the run-time system and a custom scheduler implementation specifies the operational behavior of each component. The run-time system guarantees that it will periodically call the `dispatch` function for each active scheduler, both when the processor has work in its local FIFO queue, and when it is idle. The latter call can be used by the scheduler implementation to request more work from its counterparts on other processors. The scheduler implementation in turn is required to process threads in the lower levels of the scheduling hierarchy every time it dispatches a thread from its queue.

The other aspect of the contract pertains to quiescence processing. The run-time system detects quiescence when all FIFO queues and custom schedulers in the system are idle. The run-time system invokes the `quiescence_callback` function for each active custom scheduler whenever quiescence is detected. Thus, flexible scheduling strategies can be set up across program phases with different sets of schedulers being active in different phases.

6.5 Library of Load-Balancing Policies

In this section, we describe a collection of object placement, thread placement, and thread-ordering policies for load balancing. The choice of these policies was guided by the requirements of applications described in Chapter 7. These policies exhibit a wide range of capabilities such as locality and priority sensitivity, and their efficient implementations incur a range of costs on our target architectures. Consequently, different policies yield the best performance for different application attributes.

Table 6.1 summarizes these policies, described in detail below. For each set (object placement, thread placement, and thread ordering) we describe, in turn, the semantics of the mechanisms, their implementation, and utilization of various communication architecture interfaces. Since our primary interest is in load balancing across multiple address spaces, we restrict our attention to emulations of the MSG, PUT/GET, and LD/ST interfaces on both our target platforms.

Table 6.1 Library of load balancing policies.

CLASS	POLICY	DESCRIPTION
Object Placement	`ObjectLocation`	Placement and collocation of individual data objects.
	`GroupDistribution`	Distribution and alignment of an object collection.
Thread Placement	`WorkSharing`	Thread scheduling using a sender-initiated randomized work sharing [121] scheduler.
	`WorkSharingLocal`	Locality preserving variant of work sharing, which uses a user-supplied function to determine task distribution targets.
	`WorkStealing`	Thread scheduling using a receiver-initiated work stealing [122] scheduler.
Thread Ordering	`ObjectLocationPriority`	`ObjectLocation` augmented with a per-processor priority scheduler.
	`GroupDistributionPriority`	`GroupDistribution` augmented with a per-processor priority scheduler.
	`WorkSharingPriority`	`WorkSharing` augmented with a per-processor priority scheduler.
	`WorkSharingLocalPriority`	`WorkSharingLocal` augmented with a per-processor priority scheduler.
	`WorkStealingPriority`	`WorkStealing` augmented with a per-processor priority scheduler.
	`WorkStealingGlobalPriority`	Global priority scheduler where all tasks are collected into a centralized queue prior to distribution.

6.5.1 Object placement

Two object placement policies, `ObjectLocation`, and `GroupDistribution`, are included in the Concert system.

Semantics. `ObjectLocation` controls individual object placement and collocation. On the other hand, `GroupDistribution` distributes object collections according to several standard (e.g., block, cyclic, block-cyclic, etc.) and full custom layouts. Both mechanisms indirectly distribute threads for load balance, executing a thread local to a distinguished object that it accesses.

The `ObjectLocation` and `GroupDistribution` policies are associated with object and collection creation sites respectively, using the placement syntax with the `new` operator described in Section 6.4.1. `ObjectLocation` species the target processor for object creation, while `GroupDistribution` species the distribution strategy for an object collection.

Implementation. The implementation of `ObjectLocation` is straightforward: A request is made at runtime to allocate an object on a specific processing node.

The `GroupDistribution` mechanism is implemented on top of the default physical layout of object collections whose elements are distributed in a cyclic fashion across the processors. Regular layouts such as block, block-cyclic, etc., are realized by renaming physical collection elements to correspond to different logical elements, based on the layout. Full custom layouts require the application to supply a *distribution map*, which enumerates the distribution of collection elements across virtual processors. The implementation uses these maps to redirect access of a logical collection element to the appropriate physical element.

Use of communication architecture interfaces. Thread distribution is dictated by object placement: on thread creation, the thread arguments are sent to the target processor using an asynchronous message send. Given the efficient, decoupled nature of our messaging implementation described in Chapter 4, the communication architecture does not have a big impact on performance here. Consequently, we employ identical implementations of the `ObjectLocation` and `GroupDistribution` mechanisms across all the communication architecture interfaces.

6.5.2 Thread placement

Three thread placement policies (`WorkSharing`, `WorkSharingLocal`, and `WorkStealing`) are provided in the Concert system. All three policies explicitly control thread placement, with the latter two providing various levels of locality sensitivity.

Semantics. The `WorkSharing` mechanism provides a sender-initiated randomized work sharing scheduler [121]: on creation, threads are distributed to a randomly selected processor from the entire set. `WorkSharingLocal` provides a locality-sensitive variant, where the choice of the target processor is made according to a user-supplied hashing function applied to the arguments. Thus, `WorkSharingLocal` enhances data reuse by mapping all threads accessing the same set of objects to the same processor. `WorkStealing` provides a receiver-initiated load-balancing mechanism [122]. Threads are enqueued locally upon creation. When a scheduler becomes idle, it "steals" threads from schedulers on neighboring processors.

Implementation. Implementations of `WorkSharing` and `WorkSharingLocal` first evaluate the target processor where to distribute the thread, and then send the thread arguments there using an asynchronous message send. In the case of `WorkSharing`, evaluation consists of picking a random target, while for `WorkSharingLocal`, it involves a call to the user-supplied hash function passing the thread arguments. Threads sent to a processor are extracted from the network and enqueued into the local FIFO queue.

The `WorkStealing` policy is more involved, and utilizes the custom scheduler framework. On thread creation, a context is created for the thread and enqueued into the custom scheduler. Threads are processed in the quanta assigned to the custom scheduler. When a scheduler runs out of work, it iterates over the other processors (starting from its immediate neighbor) looking for additional work. On finding work, one or more threads are stolen, i.e., transferred from the original queue into the requester's queue. Typically, multiple threads are transferred in response to a steal request to amortize the cost of remote communication. The steal request and response protocols are implemented as message-driven coroutines.

Use of communication architecture interfaces. Since implementations of both work-sharing policies (`WorkSharing` and `WorkSharingLocal`) utilize asynchronous message sends, their performance does not improve significantly because of a richer communication architecture. Consequently, we employ identical implementations across all communication architectures.

In contrast, the `WorkStealing` implementation exhibits different characteristics based upon the underlying communication architecture. Since enqueuing and stealing tasks requires atomic updates to the scheduler queues, on the MSG and PUT/GET architectures, the implementation requires participation of both the idle processor and the victim processor to complete a steal request. On the other hand, the LD/ST architecture provides remote atomic operations, so the requesting processor can directly access threads in the victim processor's queue. Consequently, our implementation on the MSG and PUT/GET architectures uses explicit messages to send steal requests and responses, with the `WorkStealing` protocol implemented in the message handlers. This implementation has the characteristic that idle time increases if the victim processor is unresponsive to incoming requests. On LD/ST architectures, the protocol is implemented to execute entirely on the requester, which directly accesses remote scheduler state. However, because all scheduler state is subject to remote steal actions, even local enqueues must be protected with a lock.

6.5.3 Thread ordering

Six policies provide load balance while preserving thread execution priorities and are suitable for search-based symbolic applications.

Semantics. `ObjectPlacementPriority` and `GroupDistributionPriority` are variants of the object placement mechanisms augmented with a per-processor priority scheduler. Similarly augmenting the thread placement policies produces the `WorkSharingPriority`, `WorkSharingLocalPriority`, and `WorkStealingPriority` policies. The `WorkStealingGlobalPriority` policy is a work-stealing variant that enqueues all tasks at a centralized scheduler instead of the local one. A global queue permits better control over prioritization.

Implementation. Implementations of the thread-ordering policies are similar to the nonpriority object and thread placement policies described earlier. The only difference is that all threads subject to a thread-ordering policy reside in a local priority queue which is attached to the runtime system using the custom scheduler framework. The scheduler uses 64-bit integer priorities.

Use of communication architecture interfaces. Similar to object and thread placement policies, the `WorkStealingPriority` and `WorkStealingGlobalPriority` mechanisms benefit more from a LD/ST interface than from a MSG or PUT/GET interface. Implementations of all other policies are identical across different architectures.

6.5.4 Summary

Our framework provides three kinds of load balancing policies: object placement, thread placement, and thread ordering. These policies represent a range of costs and sensitivity to locality and priority across various communication architectures. In the next section, we use a task generation microkernel to characterize for various application archetypes, the suitable policies.

6.6 Relating Application Archetypes to Policies

In this section, we characterize suitability of the load balancing policies for irregular applications with different attributes. We base our evaluation upon a synthetic task generation microkernel that allows the convenient modeling of various thread structure and data access characteristics. Our objective is to qualitatively identify, for various application archetypes, a set of load balancing policies which achieve high performance.

This section is organized as follows. Section 6.6.1 informally describes the space of irregular applications in terms of attributes which affect load balance. Section 6.6.2 introduces a synthetic microkernel that conveniently models applications exhibiting a range of these attributes. Section 6.6.4 provides an overview of experiments which characterize the performance of different load balancing policies for different microkernel configurations. Finally, we summarize the performance of each policy in Section 6.6.5, identifying for various application archetypes, a set of suitable load balancing policies. The performance issues highlighted by these experiments will be corroborated later for whole application programs in Chapter 7.

6.6.1 Attributes of irregular applications

In this section we informally describe the space of irregular applications in terms of a small set of attributes. Application archetypes can now be identified in terms of qualitative ranges for each of the attributes. Such a description is required before we can discuss the effectiveness of individual load balancing policies for different application archetypes.

Four attributes of the thread creation structure are of interest:

1. *Thread imbalance* refers to nonuniformity in the number of threads that are dynamically created at each creation site. Along with variations in thread granularity, this provides the main motivation for dynamic load balancing.

2. *Thread granularity* refers to the computation time of a thread, assuming all data objects are locally accessible. Both the mean thread granularity, as well as its variance across the set of threads are of interest. The former determines the overhead of load balancing policies which can be efficiently supported, while the latter provides a motivation for dynamic load balancing.

3. *Locality sensitivity* refers to the costs incurred for accessing remote data objects, and is influenced by several factors such as the initial placement of objects, and the extent of reuse between threads. Applications that have high locality sensitivity place stricter constraints on the load balancing policy which has to ensure that the advantages of a balanced load are not nullified by the cost of accessing remote data.

4. *Priority sensitivity* refers to the costs incurred for executing threads out of priority order, and is very application specific. Applications with low priority sensitivity do not care about relative thread execution order, while applications with high priority sensitivity result in significant amounts of additional computation if priorities are not enforced.

Different application archetypes exhibit different qualitative ranges of these parameters. For example a regular application which does not require any dynamic load balancing policies may exhibit low thread imbalance, uniform and large thread granularities, and low degrees of locality and priority sensitivity. A challenging application may exhibit a high degree of thread imbalance, wide variations in thread granularity, and high degrees of locality and priority sensitivity. Thus, the effectiveness of a load balancing policy is closely dependent on the application archetype.

6.6.2 A task generation microkernel

Figure 6.6 shows a synthetic microkernel that is used to model these attributes, enabling a systematic characterization of the policies described in Section 6.5.

The synthetic microkernel consists of an initial invocation of `slave_thread` on each processor. `slave_thread` has a fairly simple thread structure, computing for a while, and then conditionally creating child threads which exhibit similar behavior. In real applications, the thread computation time is determined by the thread granularity as well as the application's locality and priority sensitivities. To easily model different ranges of these attributes, we have chosen to separate these different components in the microkernel — `locality_penalty`, `compute`, and `priority_penalty`. These, and other parameters determining microkernel execution behavior are described below:

- *Thread imbalance* (line 9) models imbalance in the thread creation structure. The parameter supplies the value of standard deviation to a normal distribution with mean $\kappa = 1000$, and is used to generate the number of tasks in the task tree rooted at each zero-level thread. To keep the microkernel simple, we assume that these task trees are balanced, i.e., have the same arity value

```
1   slave_thread( int level, int id, Object *obj ) {
2       // localize object if not local
3       locality_penalty( obj );
4       // compute: penalize for out-of-order execution
5       priority_penalty( id );
6       compute( obj );
7       // create new tasks
8       if ( level < MAX_LEVELS ) {
9           arity = select_arity( id );
10          conc for ( i=0; i<arity; i++ )
11              slave_thread( level+1, select_id(id), select_object(id,obj) );
12      }
13  }
```

Figure 6.6 Code structure of the task generation microkernel. The execution behavior of the microkernel is determined by several parameters (corresponding to the boxed lines): the locality sensitivity (lines 3 and 11), the priority sensitivity (lines 5 and 11), the thread granularity (line 6), and the imbalance in thread generation (line 9).

for for each child task. We fix the value of MAX_LEVELS to 3, so the average arity value is 10 (since the mean number of tasks is $\kappa = 1000$).

- *Thread granularity* (line 6) models the actual thread computation time assuming all data are local. In practice, the thread granularity varies across threads, but to keep the microkernel simple we assume that all threads in the computation have the same granularity value. This parameter is used as a scaling factor for the application: Values of other parameters are defined as multiples of the thread granularity.

- *Locality penalty* (lines 3 and 11) models the locality sensitivity of the thread structure. In practice, locality sensitivity depends upon both the amount of shared state accessed by a thread, and the nature of the sharing (i.e., are other threads also working against the same state). To keep the microkernel simple, we assume that each thread accesses only a single object in a read-only fashion, with the locality penalty parameter specifying the access cost. Although this simple model may appear restrictive, the more complicated aspects such as multiple objects, coherence overheads, and object concurrency control overheads can all be modeled simply by choosing a higher value of locality penalty.

Our modeling of locality sensitivity penalizes a thread if it accesses an uncached remote object. To separate data locality considerations from our characterization of the load balancing policies, we model an infinite cache on each processor. The *select_object* function in line 11 controls the association between threads and objects. A thread is associated with a random object (assuming a reuse factor of 5:1) on a processor determined by the thread's zero-level ancestor. Thus, if every thread executes on the processor where it was created, no locality penalties are incurred.

- *Priority penalty* (lines 5 and 11) models the priority sensitivity of the thread structure. In practice, priority sensitivity is application-specific, but typically manifests itself as redundant work in the computation. To keep the microkernel simple, we charge a fixed penalty whenever a higher priority thread executes after a lower priority thread. The charge models redundant work which must be performed because the lower priority thread executed out of order. The priority penalty parameter specifies the fixed charge.

  Thread priorities are assigned at creation time (line 11) to be the rank of the processor where the parent thread is executing. Though this choice may appear strange, it permits us to assign a range of priority values to threads. Having a wide range allows us to model both low and high degrees of priority sensitivity entirely in terms of the penalty cost in line 5. Low and high values of this parameter model low and high degrees of sensitivity respectively.

The microkernel can be used to model a variety of application scenarios by supplying appropriate values to the above four parameters. The thread imbalance parameter introduces irregularity into the computation, and motivates the need for dynamic load balancing. Although the application of most of the policies described Section 6.5 in the context of this microkernel is fairly straightforward, additional clarification may be required to describe the effect of `ObjectPlacement` and `WorkSharingLocal`. The former executes each thread on the processor where the `obj` object is located. Because of the way threads are associated with objects in the microkernel, threads end up executing on the processor that they were created on. On the other hand, `WorkSharingLocal` uses a hash function to determine the target of `slave_thread` which ensures that threads using the same `obj` are mapped to the same processor.

6.6.3 Overview of experiments

We evaluate the suitability of load-balancing policies for various application archetypes by three sets of experiments which measure their sensitivity to thread granularity and to locality and priority penalties, respectively. These experiments exercise the task generation microkernel over the parameter values listed in Table 6.2. We describe the experiments in detail below.

Table 6.2 Values of task generation microkernel parameters.

PARAMETER	VALUES
Thread imbalance (v)	0.0, 0.5κ, κ
Thread granularity (τ)	$50.0\,\mu$s, $100.0\,\mu$s, $500.0\,\mu$s
Locality penalty (l)	0.0, 0.5τ, τ
Priority penalty (p)	0.0, τ, 5τ

Sensitivity to thread granularity. These experiments evaluate which policies yield good performance over different ranges of the thread granularity as different amounts of imbalance is created using the thread imbalance parameter. Measurements are reported only for the object placement and thread placement policies. The specific experiments measure performance of the following microkernel configurations (the parameter symbols and values are from Table 6.2):

- Thread granularity (τ) is varied fixing other parameters: $\mathsf{v} = 0.0$, $\mathsf{l} = 0.5\tau$, $\mathsf{p} = 0.0$.
- Thread granularity (τ) is varied fixing other parameters: $\mathsf{v} = 0.5\kappa$, $\mathsf{l} = 0.5\tau$, $\mathsf{p} = 0.0$.
- Thread granularity (τ) is varied fixing other parameters: $\mathsf{v} = \kappa$, $\mathsf{l} = 0.5\tau$, $\mathsf{p} = 0.0$.

The three sets measure granularity sensitivity for applications that are well balanced ($\mathsf{v} = 0.0$), and those that exhibit medium ($\mathsf{v} = 0.5\kappa$) and high ($\mathsf{v} = \kappa$) degrees of imbalance.

Sensitivity to thread locality. These experiments measure the effectiveness of a load balancing policy for preserving locality while distributing threads. As before, measurements are reported only for the object placement and thread placement policies. The specific experiments correspond to the following microkernel configurations:

- Locality penalty (l) is varied fixing other parameters: $\mathsf{v} = 0.5\kappa$, $\tau = 500.0\mu$s, $\mathsf{p} = 0.0$.

The different values of locality penalty model applications which exhibit low (l = 0.0), medium (l = 0.5τ, and high (l = τ) thread locality. The higher the thread locality, the more important is locality sensitivity in a load balancing policy.

Sensitivity to thread priority. These experiments measure the effectiveness of a load balancing policy for preserving priorities while distributing threads. Measurements are reported only for the thread ordering policies. The specific experiments correspond to the following microkernel configurations:

- Priority penalty (p) is varied fixing other parameters: $\mathsf{v} = 0.5\kappa$, $\tau = 500.0\mu$s, $\mathsf{l} = 0.5\tau$.

The different values of the priority penalty model applications which exhibit a low (p = 0.0), medium ($\mathsf{p} = \tau$), and high ($\mathsf{p} = 5\tau$) need for preserving thread execution priorities.

In the next section, we describe performance achieved by different load-balancing policies for the above microkernel configurations.

6.6.4 Microkernel performance results

We characterize the performance of different load balancing policies across various communication architectures, for a range of microkernel configurations. Measurements correspond to emulations of the MSG, PUT/GET, and LD/ST architectures on the two target platforms, the Cray T3D and the SGI Origin 2000. All experiments were run on 32-node configurations.

Figures 6.7–6.12 show microkernel performance for the experiments described above. Each figure comprises six graphs, one for each emulation of the MSG, PUT/GET, and LD/ST communication architectures on the two target platforms. Each graph shows the average microkernel execution time, normalized with respect to the the time spent on actual computation. For Figures 6.7–6.11, the normalized execution time is split into three components: *compute*, *overhead*, and *idle*. The overhead portion reflects the localization penalties and the costs of using the load balancing policy. Figure 6.12 shows an additional component—*extra work*—which reflects the time spent on redundant computation because of executing threads out of order.

We discuss in turn, the performance sensitivity of-load balancing policies to thread granularity (Section 6.6.4.1), thread locality (Section 6.6.4.2), and thread priority(Section 6.6.4.3). In each section, we first discuss performance characteristics of individual policies, and then the impact of various communication architectures.

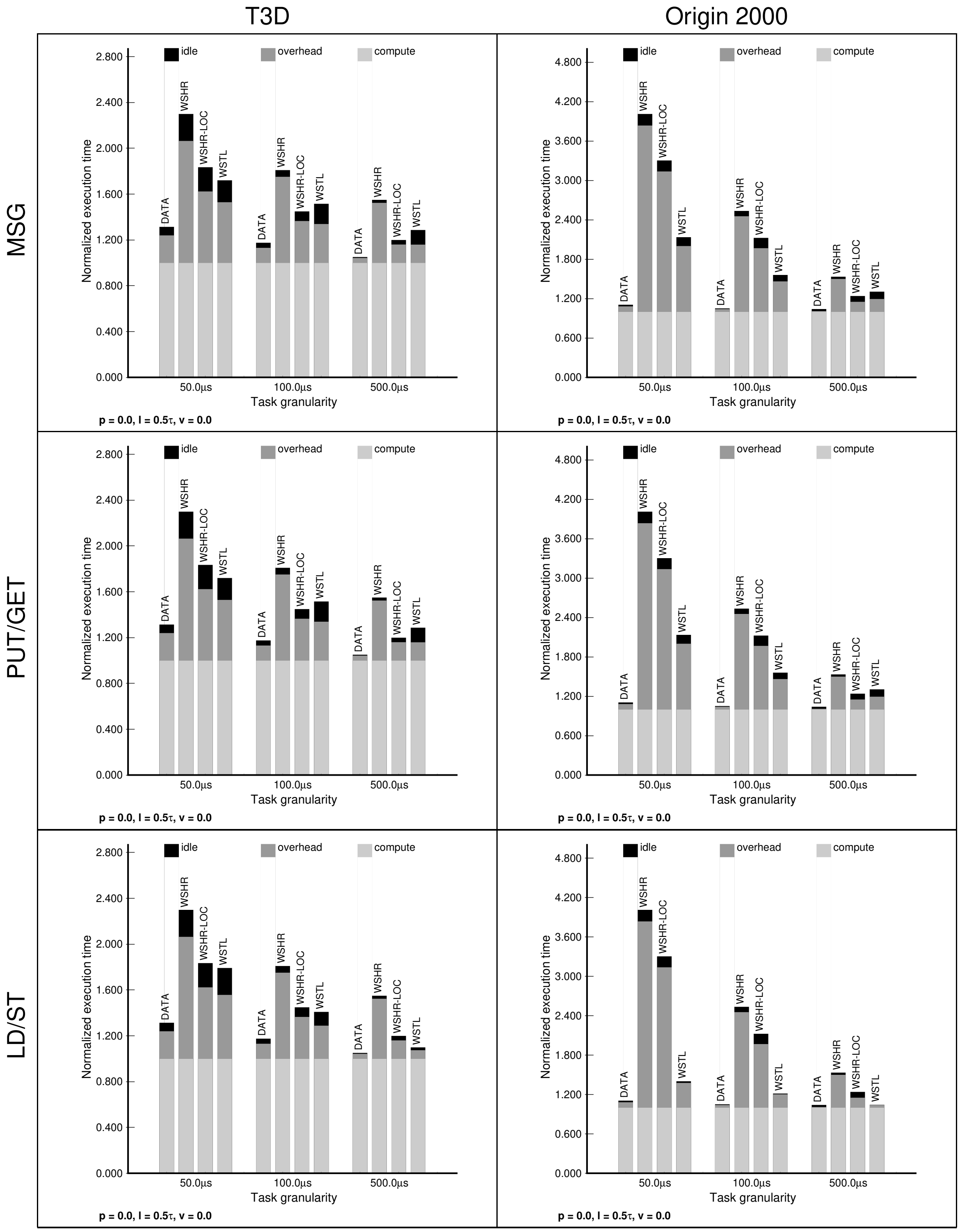

Figure 6.7 Microkernel execution time breakdown as the thread granularity is varied with no thread imbalance (ν = 0.0).

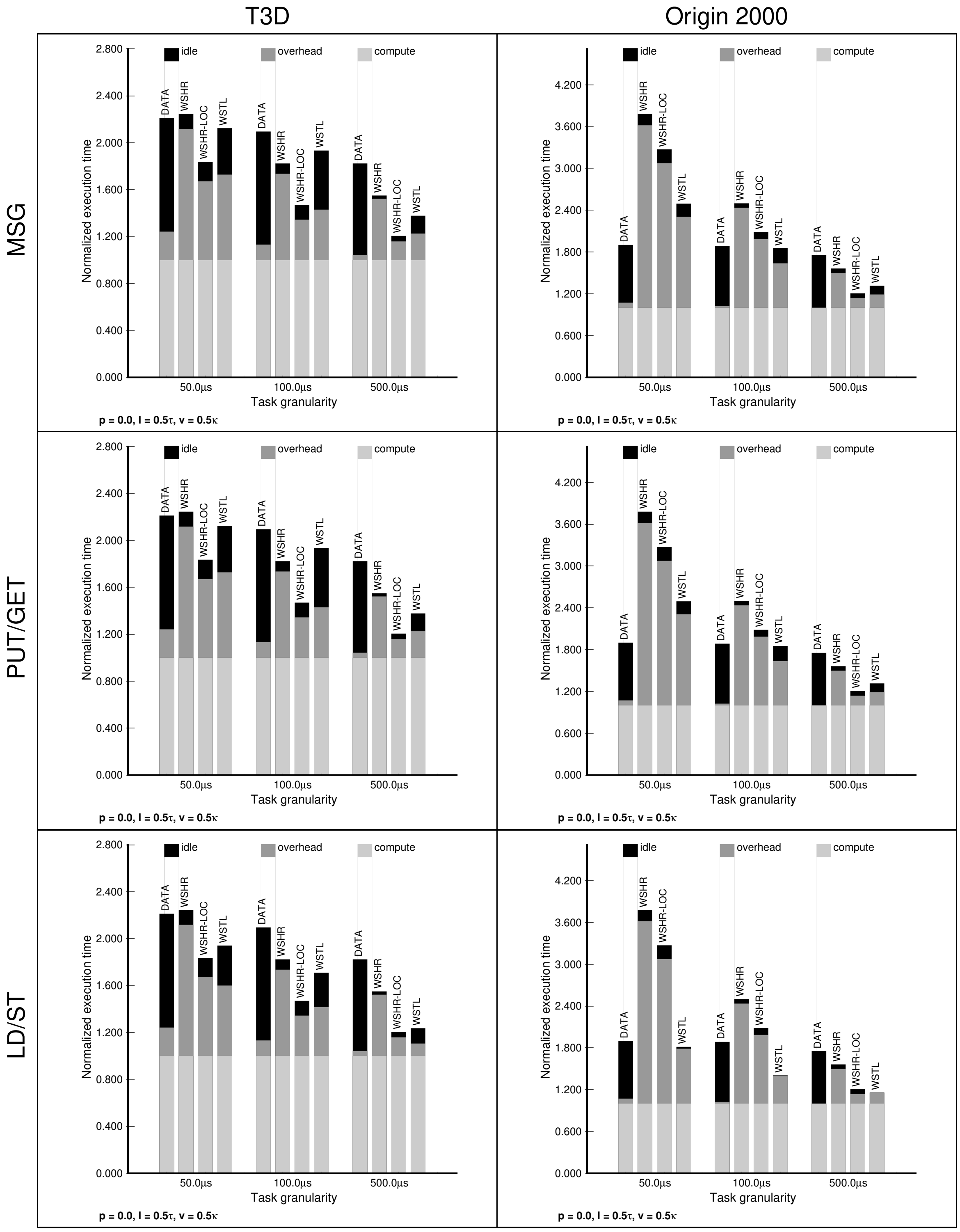

Figure 6.8 Microkernel execution time breakdown as the thread granularity is varied with medium thread imbalance ($v = 0.5\kappa$).

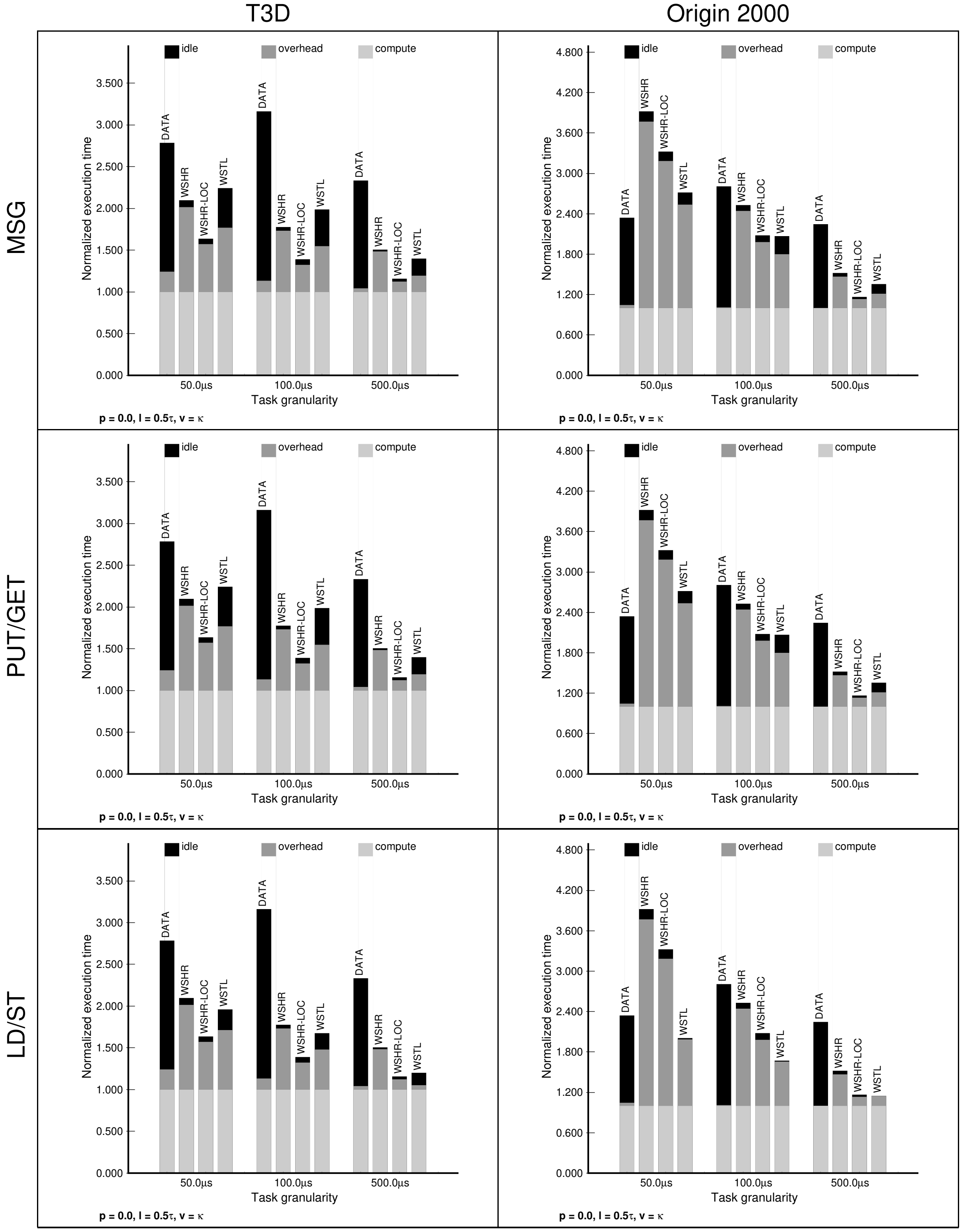

Figure 6.9 Microkernel execution time breakdown as the thread granularity is varied with high thread imbalance ($v = \kappa$).

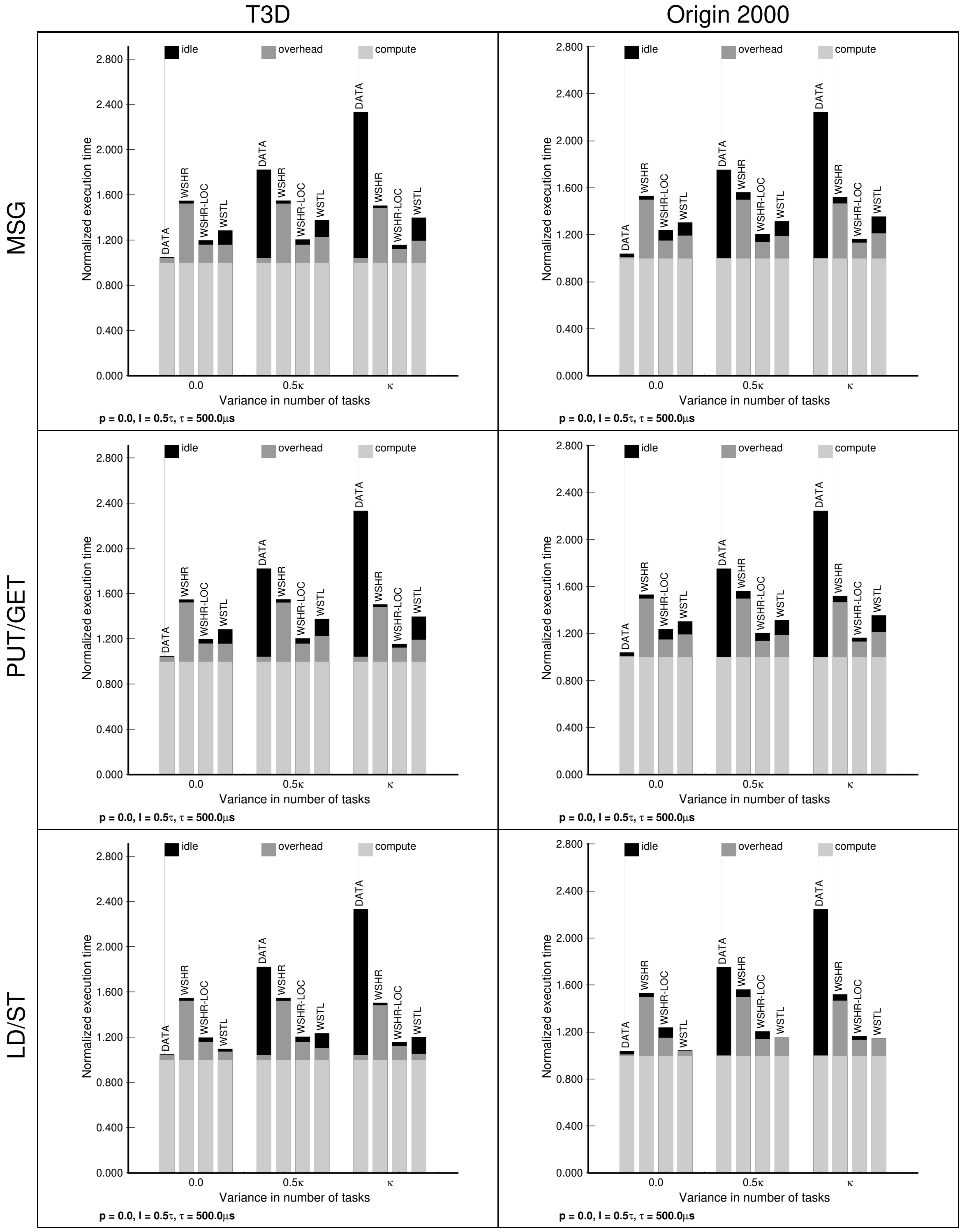

Figure 6.10 Microkernel execution time breakdown as the thread imbalance is varied.

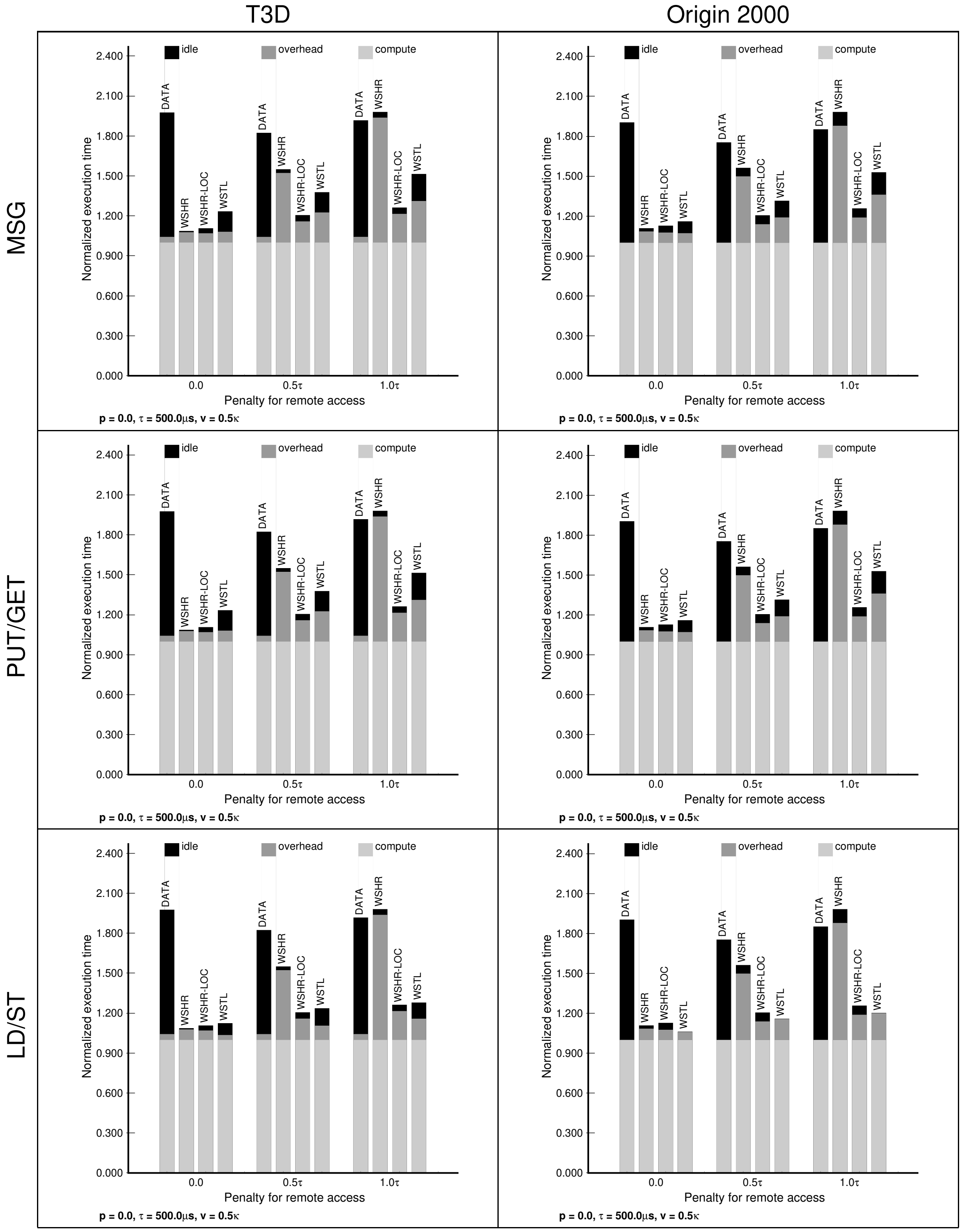

Figure 6.11 Microkernel execution time breakdown as the locality penalty is varied.

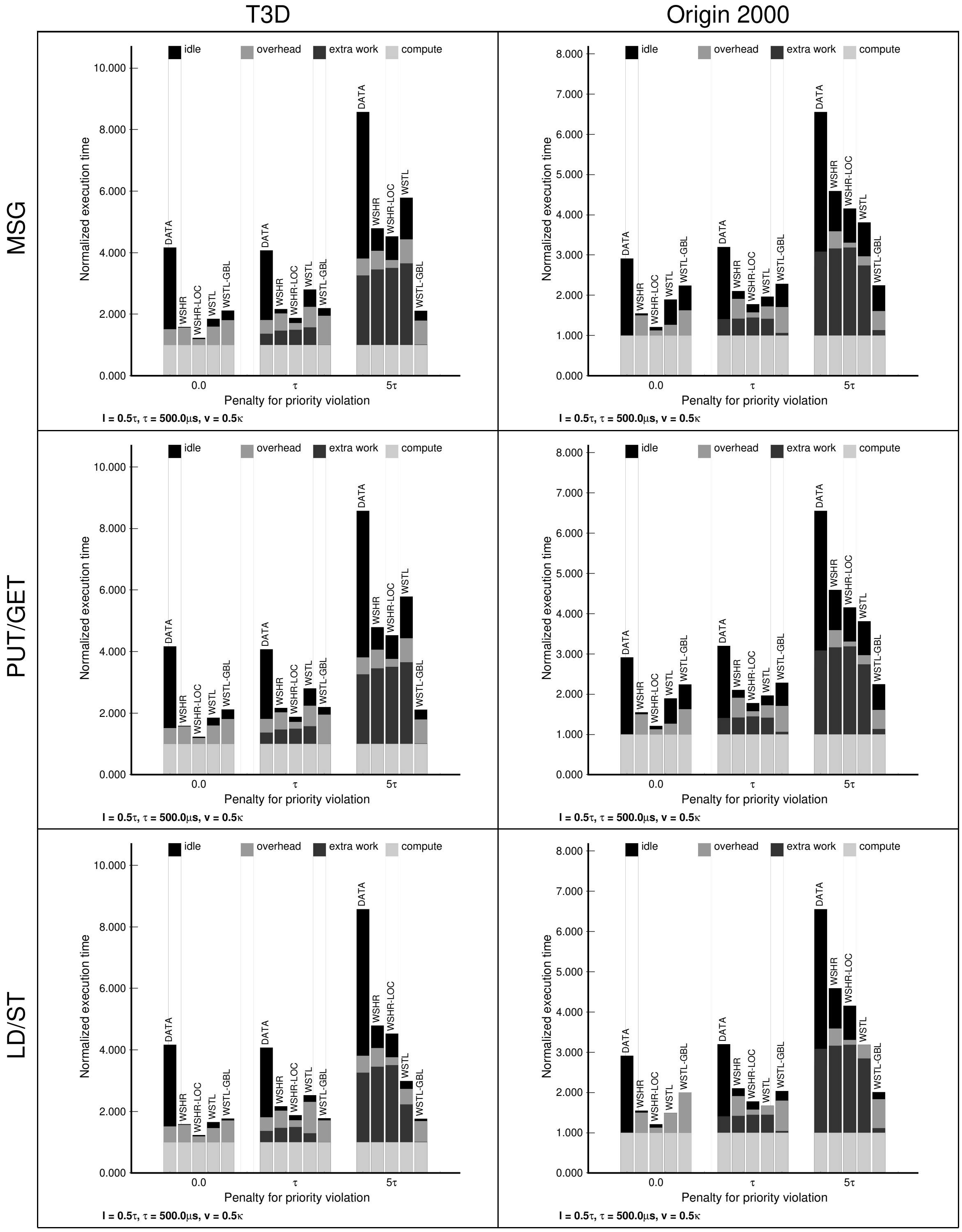

Figure 6.12 Microkernel execution time breakdown as the priority penalty is varied.

6.6.4.1 Sensitivity to thread granularity

Figure 6.7 shows the breakdown of microkernel execution time as thread granularity is varied with no thread imbalance and other parameters fixed as described above. Figures 6.8 and 6.9 show the corresponding graphs for medium and high thread imbalance respectively. Each of the graphs shows normalized execution times for four load balancing policies: DATA, WSHR, WSHR-LOC, and WSTL denote the `ObjectLocation`, `WorkSharing`, `WorkSharingLocal`, and `WorkStealing` policies respectively.

The following observations can be made about the characteristics of load balancing policies and their use of different communication architectures:

Characteristics of policies. When there is no thread imbalance (Figure 6.7), i.e., tasks are perfectly balanced across the processors, the performance reflects the base overheads of each policy:

- DATA incurs the least cost, requiring only the default run-time action of enqueuing and dequeuing threads from the FIFO scheduler.

- WSHR incurs the highest base costs across all microkernel configurations because the policy distributes threads eagerly, and without regard to locality: These costs are reflected in the overhead component of execution time.

- WSHR-LOC reflects the benefits of adding locality sensitivity to the WSHR policy: For the microkernel configurations, performance improves by up to 30%.

- WSTL incurs costs higher than DATA, but lower than WSHR. These costs reflect the overhead of enqueuing threads into the custom scheduler queue (to enable other schedulers to steal work), and are lower than WSHR because the balanced configuration implies that threads are rarely, if ever stolen. Thread execution incurs low localization penalties, pointing out the better locality sensitivity of work stealing policies.

When the microkernel exhibits varying degrees of thread imbalance (Figures 6.8 and 6.9), performance of individual policies reflects their load balancing capabilities:

- DATA achieves good performance for small thread granularities but delivers the worst performance for higher thread granularities. At small granularities, lower base costs of DATA are able to outperform higher cost policies, particularly on the Origin. On the other hand when granularity is large, thread imbalance results in increased idle time, degrading performance. Thus, the performance of the DATA policy is effective only to the extent that a good data placement ensures a balanced thread distribution.

- WSHR is more effective at load balancing, delivering better performance at granularities which are higher than a threshold that can compensate for its base costs. This threshold granularity decreases with an increase in thread imbalance. A closer examination of the execution time breakdown reveals an interesting trend: the performance of WSHR is limited by the overhead component which reflects the policy's disregard for thread locality. Thus, WSHR is likely to outperform other policies if application threads exhibit low sensitivity to locality.

- WSHR-LOC achieves performance better than the other policies because it is able to ensure load balance while preserving locality. However, as with DATA, the performance of the WSHR-LOC policy is effective only to the extent that the hash function ensures a balanced thread distribution.

- WSTL delivers roughly comparable performance to WSHR, except on the LD/ST communication architecture where it is much better. The reasons for this are discussed below. On the MSG and PUT/GET architectures, performance is limited by idle time. This highlights two aspects of the WSTL policy. First, its implementations require responsive processing of steal requests. Second, the policy exhibits poor scaling when task distribution is skewed (see Figure 6.9).

Use of communication architectures. Only the WSTL policy exhibits performance characteristics that depend upon the underlying communication architecture. It is able to take advantage of the LD/ST architecture to deliver the best performance across all microkernel configurations. The LD/ST architecture permits work stealing to be accomplished without involving the victim processor: consequently, it exhibits better scaling, idle time is lower than on the other architectures and performance is higher.

Figure 6.10 shows the data in Figures 6.7–6.9 from another perspective: the graphs show how each policy performs for variations in thread imbalance as other parameters are kept fixed.

The graphs show that the DATA policy performs best when controlling thread distribution via object placement results in low thread imbalance. When thread imbalance increases, the other policies WSHR, WSHR-LOC, and WSTL are more effective. WSHR-LOC delivers the best performance as long as the user-supplied hash function ensures a balanced distribution. WSHR and WSTL yield comparable performance, but are limited by different factors: The performance of WSHR policies is determined by locality sensitivity of threads, whereas WSTL is limited by unresponsiveness of processors either because it is busy processing or due to contention resulting from a skewed task distribution. The latter problem is addressed by a richer communication architecture such as LD/ST, which permits a responsive implementation of WSTL that delivers best overall performance.

6.6.4.2 Sensitivity to thread locality

Figure 6.11 shows the breakdown of microkernel execution time as the locality penalty is varied with the other parameters fixed as described earlier. As before, the graphs show normalized execution times for the DATA, WSHR, WSHR-LOC and WSTL policies.

Characteristics of policies. The graphs show the sensitivity of different load balancing policies to thread locality. When thread locality is low, i.e., threads incur low overheads for accessing shared state, the WSHR policy performs the best. However, when locality penalties increase, the performance of WSHR-LOC and WSTL policies are better because of their locality sensitivity. As stated earlier, the performance of WSHR-LOC is limited by the extent to which the user-supplied hash function results in a balanced thread distribution. WSTL provides more robust behavior but incurs additional idle time on MSG and PUT/GET communication architectures because of poor responsiveness to steal requests.

Use of communication architectures. Only the WSTL policy exhibits different performance characteristics across the different communication architectures. The LD/ST architecture enables a responsive implementation of WSTL, which reduces idle time incurred with the MSG and PUT/GET architectures. WSTL on the LD/ST architecture delivers the best performance when thread locality is high.

6.6.4.3 Sensitivity to thread priority

Figure 6.11 shows the breakdown of microkernel execution time as the priority penalty is varied with the other parameters fixed as described earlier. Each graph shows normalized execution times for five thread-ordering policies: DATA, WSHR, WSHR-LOC, WSTL, and WSTL-GBL denote the `ObjectLocationPriority`, `WorkSharingPriority`, `WorkSharingLocalPriority`, `WorkStealingPriority`, and `WorkStealingGlobalPriority` policies.

The following observations can be made about the characteristics of different load balancing policies, and their use of various communication architectures.

Characteristics of policies. The graphs show how effective each policy is at preserving execution priorities while distributing threads for load balance.

When there is a low penalty for violating thread execution order, the policies perform as described earlier: DATA works well for small thread granularities, WSHR works well when thread locality is low, and WSHR-LOC and WSTL work well with higher thread locality. WSTL-GBL does not scale as well as the other policies because it incurs overheads for enqueuing and dequeuing tasks from the central queue.

However, when there is a higher penalty for violating thread execution order, the policies can be classified into three categories. The first category consists of the DATA policy, which shows significantly degraded performance both due to its inability to cope with thread imbalance, and the extra work that it needs to do because of violating priority order. The WSHR, WSHR-LOC, and WSTL policies constitute the second category and yield comparable performance. All the mechanisms are more effective at dealing with thread imbalance but only preserve thread execution order locally. The last category consists of the WSTL-GBL policy which preserves global thread execution order and hence yields the highest performance for high priority penalties. However, the base costs of using the WSTL-GBL mechanism are higher, requiring a higher threshold thread granularity.

Use of communication architectures. Only the WSTL and WSTL-GBL mechanisms exhibit performance that differs across various communication architectures. Both mechanisms take advantage of the LD/ST architecture to reduce idle time incurred by delays in the processing of steal requests.

6.6.5 Suitability of load-balancing policies

The performance results in the previous section characterize the effectiveness of a load-balancing policy for different ranges of attribute values. The qualitative trends exhibited in these results can be summarized as shown in Figure 6.13. The figure shows the four-dimensional nature of the application space, and identifies for different regions of the irregular application space, the load balancing policies which yield the best performance. Regions of the application space that are not shaded indicate the absence of a clearly superior load-balancing policy; all policies are likely to yield similar performance.

The figure indicates that there is no single policy that yields the best performance across all regions of the application space. This supports an approach such as ours, which allows each set of threads that behaves differently to be subject to a different load-balancing policy. The following observations aid the selection of a suitable policy for each set of threads:

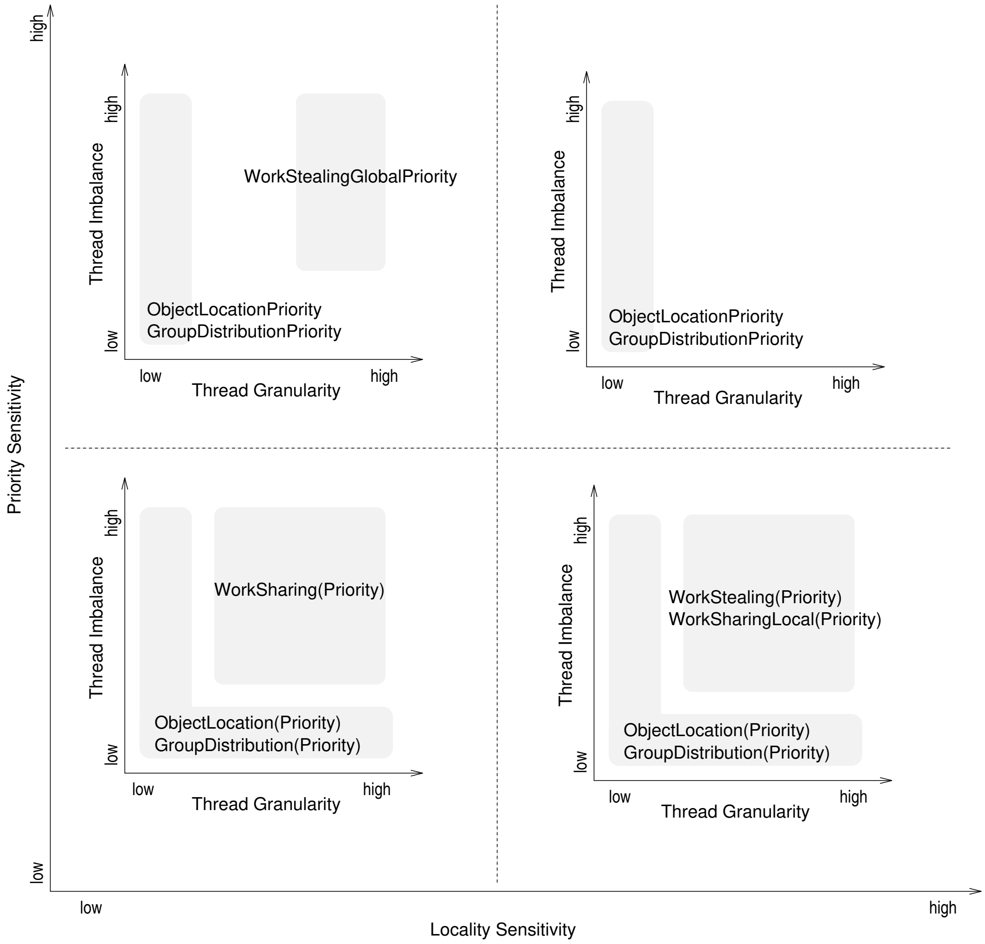

Figure 6.13 Suitability of load-balancing policies for different regions of the irregular application space.

- For applications which exhibit either very small thread granularities, object-placement policies (`ObjectLocation(Priority)`, `GroupDistribution(Priority)`) achieve best performance. In such cases, higher overhead policies result in poor execution efficiency, which outweighs any load balancing advantages.

- For applications which exhibit low thread imbalance and low sensitivity to prioritized thread execution, again load balancing policies based on object placement achieve best performance. The full generality of dynamic load balancing strategies is not required for these applications. However, it must be noted that these policies are effective only to the extent that threads access data objects in a balanced fashion.

- For applications with higher granularities which require only low sensitivity to preserving locality and prioritized thread execution, we achieve the best performance with thread placement policies based upon randomized work sharing (`WorkSharing(Priority)`). This policy incurs lower overheads for thread distribution than both `WorkSharingLocal` and `WorkStealing`.

- For applications which require low sensitivity to prioritized thread execution but where locality sensitivity is high, locality-aware policies such as `WorkStealing(Priority)` and `WorkSharingLocal(Priority)` are likely to yield better performance. Between these two policies, the choice is a function of the underlying communication architecture. On MSG and PUT/GET architectures, `WorkStealing` policies suffer from poor scaling and require increased processor responsiveness to steal requests. The availability of a LD/ST architecture eliminates these problems. On the other hand, the performance of `WorkSharingLocal` is limited by the extent to which the user-supplied hash function results in a balanced distribution.

- Finally, when applications exhibit higher sensitivity to prioritized execution, the choice of a suitable load-balancing policy depends upon the application-specific nature of priorities and how they relate to the thread creation structure. When priorities are nonuniformly distributed across the threads and the cost of violating prioritized execution is high, policies which use a centralized queue, `WorkStealingGlobalPriority` are required to collect and distribute the higher priority threads. On the other hand, when priorities are uniformly distributed across the threads (i.e., there are a sufficient number of higher priority threads), all of the thread ordering mechanisms yield good performance.

6.7 Related Work

Our research has developed a general load-balancing framework for dynamic multithreaded computations which combines run-time object placement and thread placement heuristics to efficiently distribute threads with wide variations in granularity, while preserving data locality and application execution. Related work on load-balancing techniques can be classified into three categories: theoretical results, load balancing policies, and extensible scheduling and load balancing frameworks.

Theoretical results. Both randomized work sharing [121] and work stealing [122] strategies have been shown to be optimal to within a constant factor in a probabilistic sense (i.e., the worst-case parallel time using such an algorithm is, with high probability, at most a constant times the optimal parallel time for a given task graph). A major shortcoming of these theoretical results is that, unlike our locality-aware load-balancing formulation, they do not model locality costs in the model, assuming that all data accessed by the threads is available on all processors. This shortcoming is alleviated in more recent work [161]; however, the assumed sharing model (dag-consistency) is very restrictive and not representative of general sharing behavior encountered in multithreaded computations.

Load balancing policies. Many researchers have proposed a large number of load balancing policies based on object placement [162–166], thread placement [6, 7, 9, 10, 24, 27, 82, 117–120], and thread ordering [167], which are sensitive to varying degrees to data locality. With few exceptions, most policies are inappropriate for use in dynamic multithreaded computations because they are either too heavy-weight for use with fine-granularity threads [6, 10, 13, 117–120], completely ignore locality [7, 24], or work well only with regular computations [9, 10, 27]. In addition, a major philosophical difference of our approach relates to the fact that several of these policies have been proposed as a single unified solution for application wide load balancing. While such an approach may be appropriate for regular or medium-grained computations, our microkernel studies show that a single policy is insufficient to achieve efficient load balancing for fine-grained dynamic computations. Given this difference in perspective, the related work in this category can be viewed as addressing an orthogonal problem of load balancing a particular subset of threads. Our framework builds on such component policies to synthesize overall application load balance.

Extensible scheduling and load balancing frameworks. Most similar to our approach are systems that allow different subsets of threads to be subject to different load balancing and scheduling policies during execution. Custom scheduling is relatively common in real-time and multimedia operating systems [155–160]. The primary difference of our approach with respect to real-time and multimedia operating systems is that the integration between different schedulers is considerably more efficient, relying on non-preemptive scheduling of cooperating application threads.

On the other hand, only a handful of programming systems, such as Presto [90] and Converse [88], possess similar capabilities. While support for attaching custom schedulers in both Presto and Converse share similar objectives to our approach, there are important differences. First, the thread scheduler interface in both systems is more suitable for medium-grained threads, requiring multiple indirect function calls per scheduling operation. Second, the interfaces do not specify how processing resources are allocated to multiple custom schedulers, nor how quiescence is detected for schedulers across processing nodes. Finally, although both Presto and Converse provide mechanisms for customization, few large-scale experiments have discussed the resulting benefits. In contrast, our approach is motivated by microkernel studies and substantiated by the application results described in Chapter 7.

6.8 Summary

In this chapter, we have described a framework for load-balancing dynamic multithreaded computations. Based on proof of intractability of a locality-aware version of the load balancing problem, we adopt a heuristic approach that efficiently distributes threads with wide variations in granularity, while preserving data locality and application execution priorities.

To meet these conflicting goals, our approach views the application load balancing problem as the composition of several subproblems, each of which independently handles a subset of threads. The framework provides a collection of heuristic load balancing policies of varying cost and capability. Instead of dictating the use of one policy for all threads in the computation, our approach relies on a compiler or a programmer to select the most efficient mechanism suitable for different sets of threads.

The framework consists of three components—language and run-time support, a library of load-balancing policies, and a correspondence between application attributes and policies.

Language annotations associate object and thread creation sites with specific policies, and run-time support allows the individual load-balancing policies to be composed to synthesize application load balance. The run-time support consists of a custom scheduler framework that allows processing resources to be allocated among multiple load-balancing structures, each of which implements a different policy.

The library of load-balancing policies exhibits a range of costs and capabilities, allowing different threads to be controlled by different policies, based upon considerations of thread granularity, imbalance, and required sensitivity to locality and priority. Specific policies are of three kinds: object placement mechanisms which control thread distribution by controlling target object location, explicit thread placement mechanisms which permit thread distribution independent of object placement and are additionally sensitive to data locality, and thread ordering mechanisms which ensure load balance while prioritizing thread execution.

The third component of the framework identifies suitability of different policies for specific sets of application attributes by characterizing the performance of a synthetic task generation microkernel. The microkernel results show that no single policy yields the best performance across all regions of the application space, supporting an approach which allows each set of threads that behaves differently to be subject to a different load balancing policy. Our results show that object placement mechanisms are suitable when applications exhibit low imbalance or small thread granularities. For higher thread granularity and higher degrees of imbalance, thread placement policies are appropriate. Finally, when the application also requires prioritized thread execution, thread ordering mechanisms yield best overall performance.

An application based evaluation of this framework is described in Chapter 7.

CHAPTER 7

PERFORMANCE EVALUATION

In this chapter we present an application-based performance evaluation of the data locality and load balancing mechanisms described in Chapters 5 and 6, respectively. Using four large irregular applications expressed in ICC++, we describe use of the view caching and load balancing frameworks, and characterize the resulting performance. Our results demonstrate that the mechanisms result in significant performance improvements across a range of communication architectures. The mechanisms are collectively sufficient, enabling each application to achieve performance comparable to the best achievable via low-level programming means on large-scale parallel systems. In addition, each mechanism is individually effective; its absence results in a significant performance loss.

This chapter is organized as follows. Section 7.1 describes the experimental setup and structure of the application studies. Per-application descriptions of the study are presented in Sections 7.2–7.5. Section 7.6 analyses the use of the frameworks and resulting performance. Finally, Section 7.7 summarizes our contributions.

7.1 Overview

We evaluate the view caching and load balancing framework described in Chapters 5 and 6, respectively, by measuring their impact on the performance of four irregular applications. In this section, we describe in turn the experimental setup and the structure of the application study. We conclude by describing the set of programs used in the study.

7.1.1 Experimental setup

All experiments reported in this chapter were conducted on implementations of the Illinois Concert System on the Cray T3D and the SGI Cray Origin 2000. The application programs are expressed in a high-level concurrent object-oriented programming style, using the ICC++ language. These programs were compiled using the Concert compiler, and an executable produced by linking the generated C code against a target-specific run-time library. The run-time library includes the mechanisms described in Chapters 4, 5, and 6.

7.1.2 Structure of the study

To quantify the performance impact of view caching and load balancing frameworks, we contrast application performance using these frameworks against a competitive baseline. The baseline corresponds to the application performance using a run-time system configuration which includes the local efficiency mechanisms (e.g., pull messaging and hybrid stack-heap execution) described in Chapter 4, and manages data locality using an object-consistent caching system. In this configuration, program objects are mainly distributed to enhance local efficiency while ensuring adequate concurrency, and threads execute local to the target object of the corresponding method invocation.

The use of view caching and load balancing frameworks for each of the applications is explicitly specified by the programmer utilizing the language annotations described in Section 2.3.2. While the complexity of choosing the appropriate data locality or load balance policy was significant (often comparable to the application's algorithmic complexity), the orthogonal nature of these annotations permits expression of such information to be cleanly integrated into the basic program structure. For all the applications used in the study, the optimized parallel application versions exhibit identical basic program structure to their sequential counterparts, requiring changes to fewer than 5% of the source code lines (see Section 2.3.3 for an example). The reader is referred to [21] for a detailed discussion of the programming effort involved in optimizing data locality and load balancing for the application programs listed below.

For each application, performance measurements were taken for the following four configurations:

- BASE measures the best performance achievable using a baseline run-time system configuration that includes local efficiency mechanisms, an object-consistent caching system, and places objects to enhance data locality while executing threads local to the target object of the corresponding method invocation.

- BASE+DL+LB measures the best performance achievable using the view caching and load balancing frameworks on top of the baseline.

- BASE+DL measures the performance compared to BASE+DL+LB when the load balancing framework is not used.

- BASE+LB measures the performance compared to BASE+DL+LB when the view caching framework is not used.

The subtractive order of measurements reflects the fact that data locality and load balance are not orthogonal: improving load balance may reduce data locality and vice versa. By expressing the impact of each mechanism in terms of the loss in performance with respect to the BASE+DL+LB configuration when it is absent, we leave constant the use of the mechanisms in the program. This allows us to cleanly separate the individual impact of the two sets of mechanisms from their collective impact while retaining a common basis for comparison. Specifically, the improvement in performance of the BASE+DL+LB configuration over the BASE configuration represents the *collective impact* of the view caching and load balancing frameworks. The loss in performance for the BASE+DL and BASE+LB configurations as compared to the BASE+DL+LB configuration represents the *individual impact* of the load balancing and view caching frameworks respectively.

7.1.3 Application suite

Because dynamic multithreaded computation models were proposed to manage complex parallel applications, our evaluation uses a suite of challenging irregular programs, each of which is irregular in data structure, parallel control structure, concurrency, or all three. Table 7.1 lists these applications, which span a variety of computational domains. All the applications rely on sophisticated pointer-based data structures and efficient algorithms, and are complex to program even on sequential systems.

To understand the application structure in more detail, we examined the execution profile of each application based on a single processor run on the Cray T3D. Table 7.2 lists some key statistics about the thread and object structure which characterize the application behavior qualitatively. We briefly discuss these characteristics below.

Table 7.1 The irregular applications suite.

APPLICATION	DESCRIPTION	INPUT
IC-Cedar	protein dynamics [72] simulates protein macromolecular dynamics.	myoglobin
Radiosity	computer graphics [168] computes illumination using the hierarchical radiosity method.	room
Gröbner	computational algebra [169] computes Gröbner basis.	ae-4
Phylogeny	evolutionary history [170] determines the evolutionary history of a set of species.	prim.40

Table 7.2 Sequential execution profile of the ICC++ applications on the Cray T3D. All applications involve dynamic thread and object creation, and span a range of thread granularities, object sizes, and priority sensitivities.

APPLICATION	TIME	THREAD STRUCTURE			OBJECT STRUCTURE	
	(sec)	Number	Size (μs)	Priority	Number	Size (bytes)[a]
IC-Cedar	265.67	115250	25–3000	none	10502	24
Radiosity	379.58	531823	10–5000	none	16868	216
Gröbner	486.98	58702	25–30000	high	136	100–10000
Phylogeny	200.97	240054	25–10000	medium	1125	56

[a] The object size refers only to the shared subset of object state.

Thread structure. Each application creates a large number of threads spread across the execution. With the exception of IC-Cedar, thread creation is data-dependent and closely related to how the computation evolves, exhibiting wide variations in the number of child threads created by a particular thread. IC-Cedar exhibits more structured thread creation with the same number of threads created at the start of each iteration. Despite these differences, in all applications, threads exhibit wide variations in granularity ranging from 25μs to a few milliseconds. Figures 7.1–7.4 show histograms of thread granularities on the Cray T3D. With the exception of Gröbner, all other applications contain a large number of small granularity threads, requiring low-overhead thread operations for efficiency. Most of the threads in the Gröbner application are relatively coarse-grained, contributing to large periods of processor unresponsiveness.

In addition, two of the applications—Gröbner and Phylogeny—require thread execution to be prioritized based on a heuristic metric. In Gröbner, priorities are unevenly distributed, with relatively few high priority threads. In contrast, the Phylogeny application is characterized by a larger number of

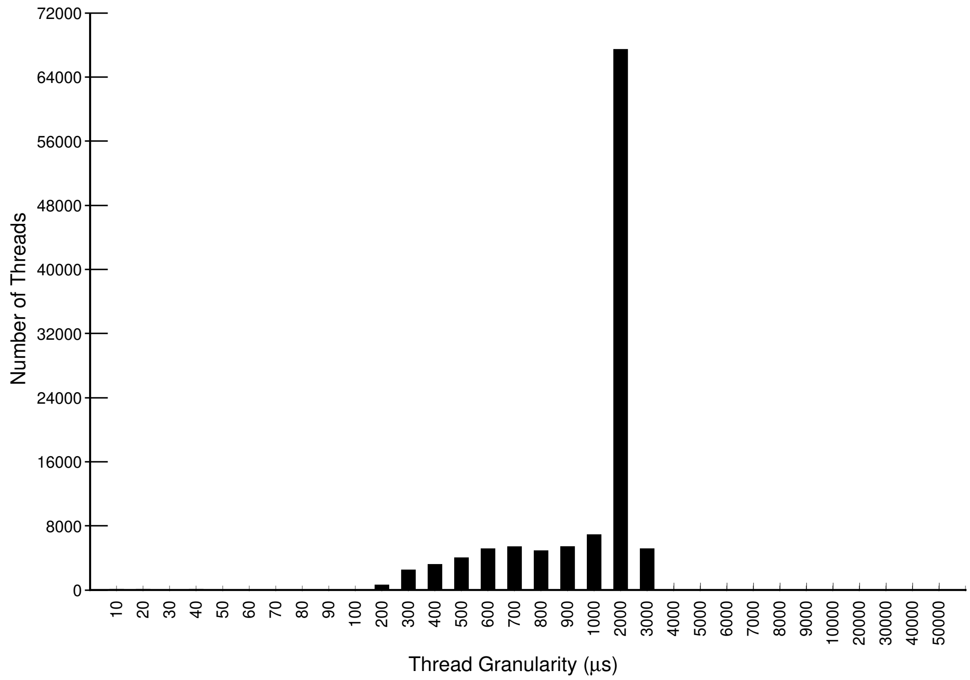

Figure 7.1 Histogram of thread granularities on the Cray T3D for the IC-Cedar application.

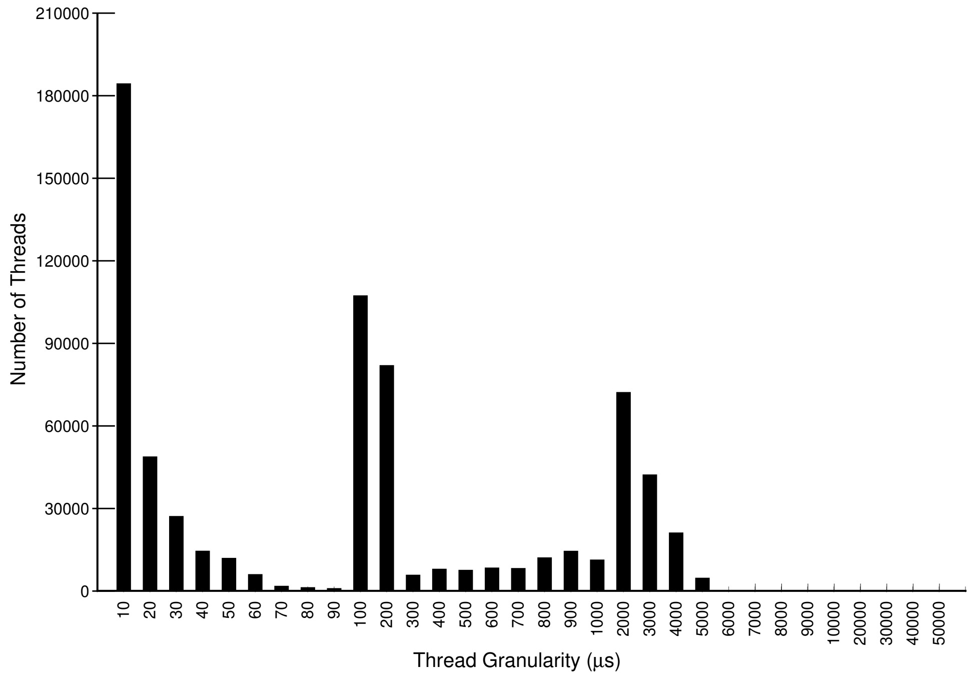

Figure 7.2 Histogram of thread granularities on the Cray T3D for the radiosity application.

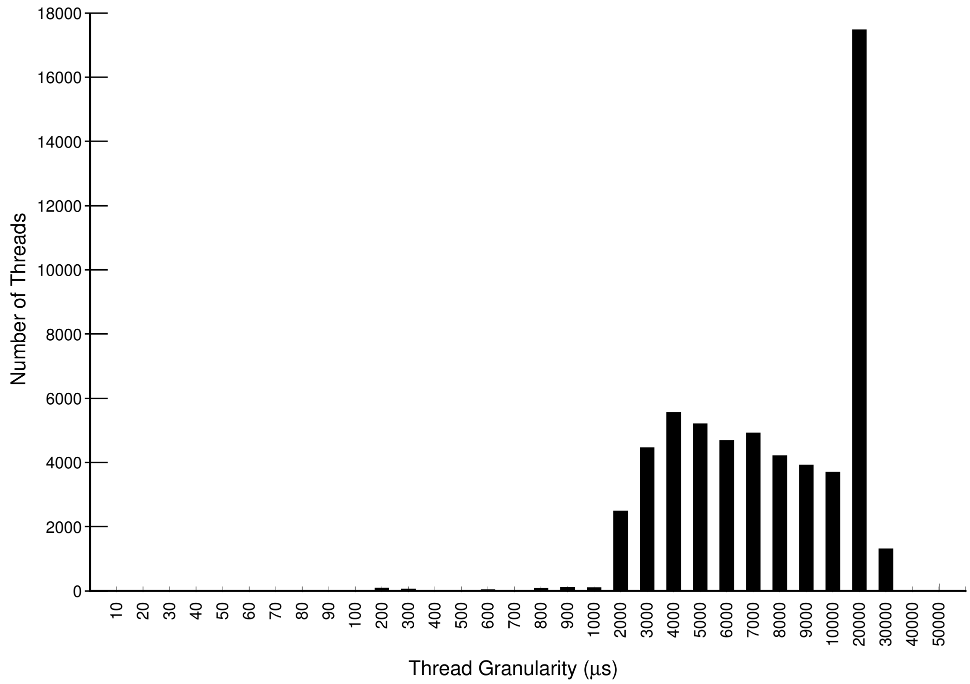

Figure 7.3 Histogram of thread granularities on the Cray T3D for the Gröbner application.

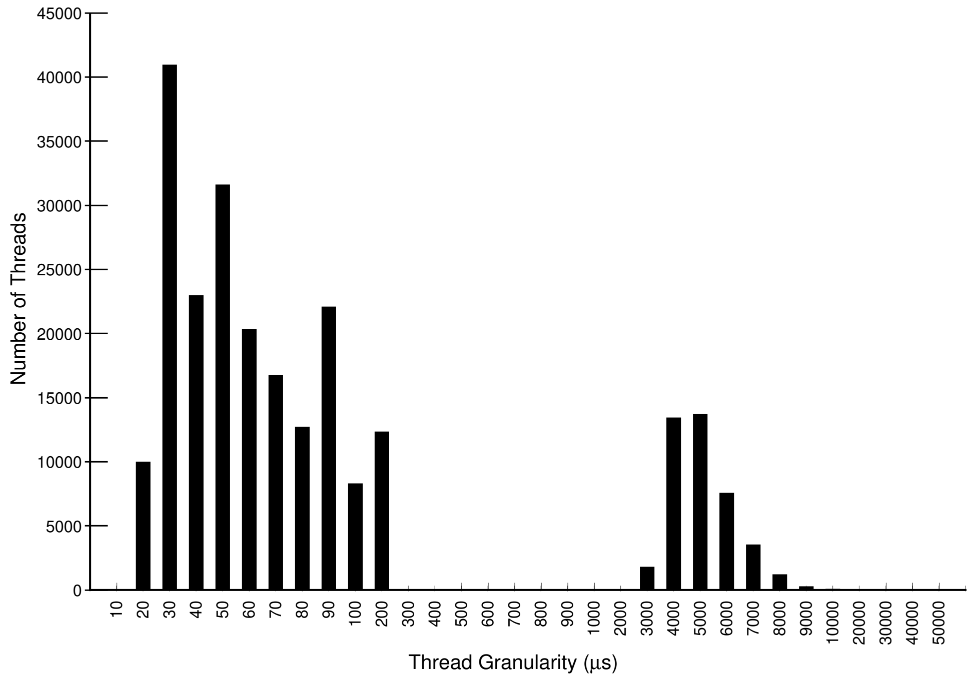

Figure 7.4 Histogram of thread granularities on the Cray T3D for the phylogeny application.

high-priority threads. Consequently, careful management of priorities is required for Gröbner, but is less critical for Phylogeny.

Thus, the thread structure of these applications shows marked differences from typical thread behavior encountered in data parallel and message passing programs. The latter exhibit far fewer threads, typically created at startup, all of which perform uniform amounts of computation. In comparison, our application threads exhibit irregularity in time (dynamic thread creation), task structure, and grain size.

Object structure. With the exception of IC-Cedar, all of the applications create a large number of objects spread across the execution. In IC-Cedar, all objects are created at startup. In each case, objects are relatively fine-grained and accessed in an unstructured fashion by the application threads. Additionally, with the exception of IC-Cedar where they stay constant, data access patterns evolve with the computation.

Thus, the object structure of these applications shows significant differences from the object structure of data parallel and message passing programs. In the latter, objects are typically created at startup, and accessed in a structured fashion by the threads. In comparison, object structure in our applications exhibits irregularity in time (dynamic object creation) and space (unstructured access).

These characteristics of thread and object structure contribute to the performance challenges described earlier in Section 3.1, making it difficult to achieve high performance for these applications on scalable parallel platforms. In fact, all previous efforts [169–172] have restricted themselves either to shared memory implementations, or have relied on complex low-level programming approaches. In the rest of this chapter, we describe the use of view caching and load balancing frameworks on each of these applications and characterize the resulting performance. As we shall see later, the run-time techniques described in Chapters 5 and 6 enable each application to achieve performance comparable to the above low-level approaches.

7.2 IC-Cedar

IC-Cedar is a parallel version of CEDAR [173], a sequential protein molecular dynamics program that models the motion of individual protein and surrounding solvent atoms using Newton's equations of motion. We describe, in turn, the application structure, its use of view caching and load balancing frameworks, and the resulting performance.

7.2.1 Application structure

Computation in IC-Cedar proceeds in discrete time steps with four computation phases: neighbor list calculation, force calculation, motion integration, and coordinate correction. The neighbor list calculation computes, for each set of protein and water atoms, a collection of interacting neighboring atoms that are within a spatial cutoff distance. The force calculation phase computes both short-range bonded forces and long-range Coulombic and Lennard-Jones forces incident on an atom. Force calculation involves pairs of atoms determined by each atom's neighbor list. Symmetry is used to update forces on both atoms (taking advantage of Newton's third law). Motion integration uses the computed forces to rescale the velocities and update atom positions according to Newton's equations of motion. Finally, the coordinate correction phase ensures that the new atom positions still preserve bond length constraints. In the discussion that follows, we restrict our attention to the force calculation phase which dominates the sequential execution time. Although this phase has ample concurrency, it is challenging to parallelize because of irregular computation and communication patterns.

The algorithm implementation uses one kind of object—atom—which contains fields for position, velocity, and force. The algorithm represents the neighbor list as a list of atom pointers, and allocates the atom objects as a collection. The latter provides a convenient means of referring to the entire group of atom objects. In each iteration, force calculation is expressed as a data-parallel operation on the atom objects. A logical thread is created to compute the forces with respect to an individual atom. Force calculation requires access to the position fields of neighboring atoms and updates their force fields. The interaction patterns are irregular given the nonuniform spatial distribution of atoms. Because the cutoff radius in typical data sets is relatively large (as compared to the overall dimensions of the simulation), a large volume of space is considered for each atom, causing irregular long-distance accesses which form a spherical spatial locality region. Figure 7.5 shows these irregular interactions between atom pairs. Finally, the force calculation threads exhibit wide variations in granularity, depending on the type of the atom (protein or water), length of each atom's neighbor list, and a number of geometric constraints (such as bond angle, bond length, etc.)

Thus, the IC-Cedar application is characterized by dynamic threads created at the start of each iteration. Although the thread creation structure is closely related to the data objects (a thread is created for each atom), these threads exhibit wide granularity variations and access a variable number of atom objects according to an irregular pattern, both reading their position fields and updating their force

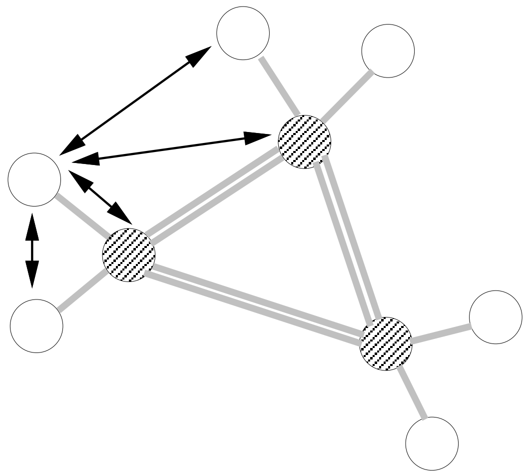

Figure 7.5 The force calculation phase of IC-Cedar involves irregular spatial interactions (shown using arrows) to access fields of neighboring atom objects.

fields. Since positions are updated every iteration (in the motion integration phase), they cannot be cached across iterations.

Performance of low-level approaches. Previous efforts [72, 171] for efficiently executing molecular dynamics simulations have relied on the inspector-executor separable communication approach [97–99]. As described in Section 3.3, these approaches are applicable for applications with irregular data access patterns that are independent of the ongoing computation. In such cases, the computation can be globally separated into two phases: the inspector (which performs all communication to retrieve positions of remote atoms and stores them in a local array) and the executor (which performs the actual computation using local arrays). This separation has several advantages which contribute to efficiency, such as permitting the grouping of multiple communication requests, eliminating per-access synchronization, and reducing cache interference between computation and communication. In previous work [72], we have shown that applying the inspector-executor approach to IC-Cedar results in a speedup of 32 on 64 nodes of the T3D on the same data set. In comparison, CHARMM [171], a highly-tuned SPMD Fortran program with calls to the CHAOS [174] run-time library, achieves a speedup of 42 on 64 nodes of the T3D for a comparable data set. Similar performance levels have also been reported on the SGI Origin 2000 for NAMD [175], a hand-tuned multilingual parallel program which combines three different paradigms: parallel message-driven objects, message passing, and coarse-grained multithreading.

7.2.2 Data locality and load-balancing optimizations

In contrast to the low-level approaches described above, our focus is on achieving good performance without changing the dynamic multithreaded expression of the program. The primary performance challenges are data locality (the cost to access remote atoms) and load balance. The BASE version of the application relies on the default cyclic distribution policy for object collections and uses an object-consistent DSM system to reduce cost of remote access. However, thread granularity variations and irregular data access patterns in IC-Cedar prevent the BASE version from achieving good performance. These shortcomings are alleviated using the view caching and load balancing frameworks as summarized in Table 7.3 and described below.

Table 7.3 Optimizing IC-Cedar performance using view caching and load balancing frameworks.

FRAMEWORK	WHERE APPLIED	POLICY CHOICES
view caching	read access to atom objects to retrieve position fields	`READ_SELF_INV` exploits application semantics that position fields get updated every iteration
load balancing	allocation of atom object collection	`GroupDistribution` specifies a custom spatial decomposition based on Peano-Hilbert encoding

Utilizing the view-caching framework. Coherence costs for caching remote atom state are reduced by exploiting application data access semantics that the object state subset (position fields) accessed by the force calculation thread is updated at the home node in each iteration. This permits the atom view to be cached using a custom coherence type, `READ_SELF_INV`. The custom protocol obtains a detached view, relying on requester-initiated `access-grant`, `data-transfer`, and `access-revoke`, and requires home node participation only for data transfer. All cached views are invalidated at the end of each iteration. Augmenting the BASE version with this custom protocol produces the BASE+DL version of the application.

Utilizing the load-balancing framework. Although it is simple to achieve load balance for IC-Cedar, not all thread distribution strategies are locality conscious. Given the thread creation structure and the large number of atoms, any balanced distribution of atoms (such as the cyclic distribution) results in good load balance. However, the default cyclic distribution does not preserve spatial locality, resulting in irregular data access patterns. Our load-balancing framework enables a custom distribution for the collection of atom objects which ensures load balance while preserving locality. We use the `GroupDistribution` policy to specify a spatial decomposition of the atom objects based on a Peano-Hilbert encoding [164, 165] traversal of spatial cells, a technique borrowed from astrophysics applications [176, 177]. As described in Section 6.5, the `GroupDistribution` policy is implemented using a distribution map supplied at startup. By controlling object placement and executing threads local to target objects, we achieve load balance while ensuring that most of the data accesses are to collocated objects. Augmenting the BASE version with the `GroupDistribution` load balancing policy produces the BASE+LB version of the application.

7.2.3 Performance results

Figure 7.6 shows the performance of the BASE, BASE+DL, BASE+LB, and BASE+DL+LB application versions for various communication architectures on the Cray T3D and the SGI Origin 2000. The graphs on the Origin also show the performance of an additional version—BASE-CC—which corresponds to the BASE version executing on top of the CC-LD/ST interface to the hardware. This version uses hardware support for cache-coherent memory operations within a single address space. The graphs show speedup with respect to the fastest single-node execution time (265.67 s on the T3D and 22.17 s on the Origin). In comparison, a sequential C++ version of the application executes in 190.56 s on the T3D and 24.20 s on the Origin. The difference in execution times on the T3D arises due to the fact that our generated code is not able to effectively utilize the small L1 cache. Measurements reported in this section are for 25 force computation iterations, and correspond to a cutoff distance of 10Å. This cutoff distance results in 1,845,523 pairs of interacting atoms for the myoglobin data set which consists of 10,502 atoms, including 1664 protein atoms and 2946 water molecules.

The general trend seen in the graphs is that view-caching and load-balancing mechanisms significantly improve performance as compared to the BASE version across the entire range of communication architectures and target platforms. The following specific observations can be made about the collective and individual impact of the mechanisms:

Collective impact of mechanisms. On the T3D, the two mechanisms contribute to the BASE+DL+LB version achieving peak speedups of 16.2, 24.8, and 25.2 on 64 nodes for the MSG, PUT/GET, and LD/ST architectures respectively. Speedups on the latter two architectures compare favorably with the speedup of 32 on 64 nodes achieved by the low-level inspector-executor technique (single-node execution times in both cases are comparable).

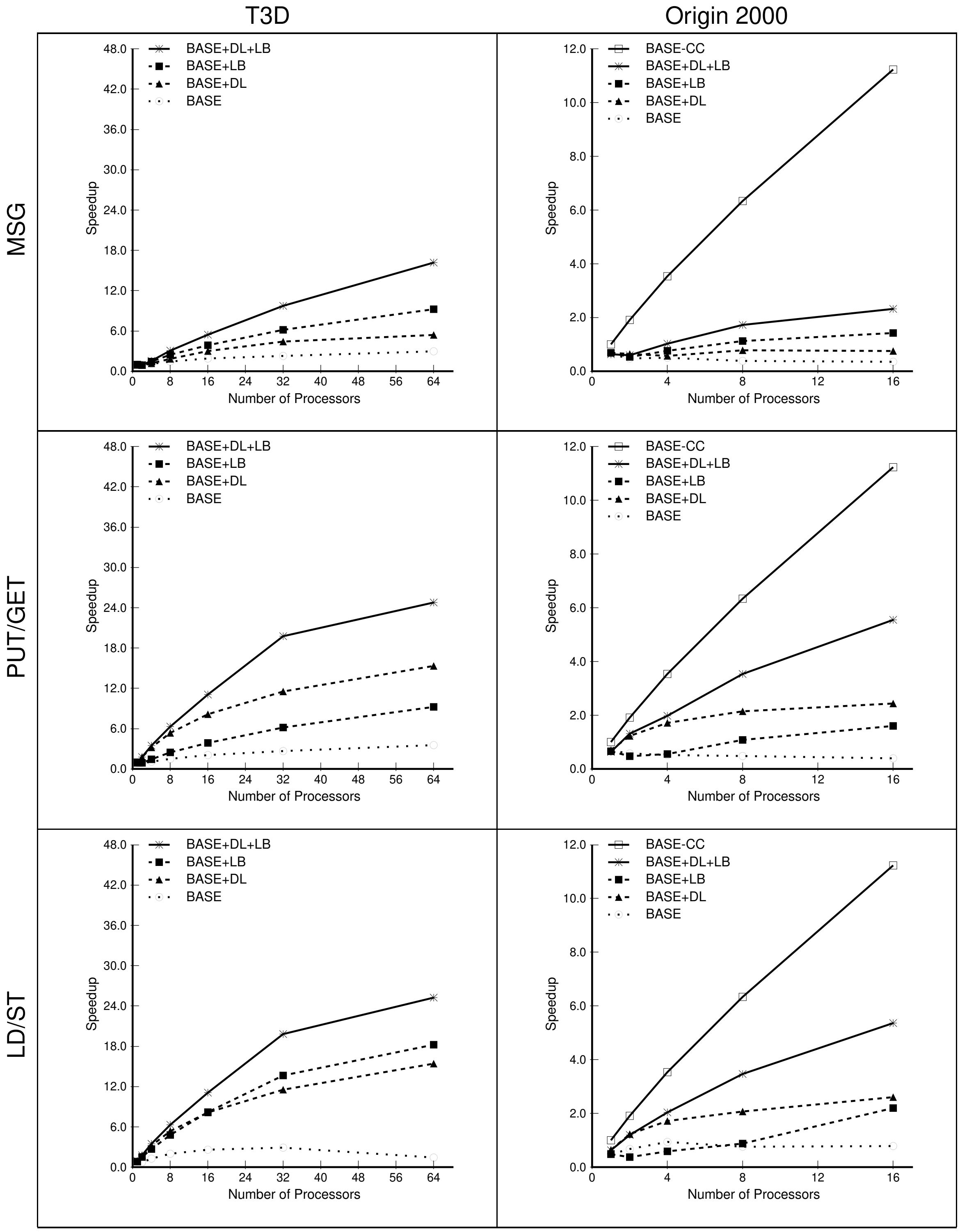

Figure 7.6 Overall performance impact of data locality and load balancing techniques for the IC-Cedar application.

On the Origin, the BASE+DL+LB version achieves peak speedups of 2.3, 5.5, and 5.4 on 16 nodes for the MSG, PUT/GET, and LD/ST architectures respectively. While these speedups represent significant improvements over the BASE version, they fall short of the speedup of 11.2 on 16 nodes achieved by the BASE-CC version which utilizes hardware support for cache-coherent memory operations. The performance difference primarily arises because of higher access control costs associated with a software implementation of object caching, relative to the thread granularity. In fact, the single-node execution time of the BASE+DL+LB version reflects a slowdown of 35% over the BASE-CC version.

Impact of view-caching framework. Comparing performance of BASE+LB and BASE+DL+LB versions, we find that not using the view caching framework decreases the achieved speedup on 64 nodes of the T3D to 9.2, 9.2, and 18.2 for the MSG, PUT/GET, and LD/ST architectures. The corresponding speedups on the Origin are limited to 1.4, 1.6, and 2.2 on 16 nodes for the three architectures.

To explain this loss in performance, we show a breakdown of the execution times for the two versions on a subset of processors in Figure 7.7. The execution time is decomposed into four components (*compute* time, run-time *overhead*, *coherence* overhead, and *idle* time) and normalized with respect to the compute time. In the figure, the label BASE+VC+LB denotes the use of view-caching mechanisms for data locality.

The data in the figure shows that using the view-caching mechanisms reduces both idle time and coherence overheads. The view-caching framework caches remote atom objects using the `READ_SELF_INV` custom coherence protocol, which requires home node involvement only on the MSG architecture for data transfer. Consequently, the custom protocol incurs reduced coherence overheads and idle time because it does not need to wait for remote processors to respond to coherence requests. The graphs also show that although this improvement is present across all architectures, it is most significant for the PUT/GET architecture. Unlike synchronous object-consistent DSM protocols, the decoupled `READ_SELF_INV` protocols are able to take advantage of one-sided hardware communication primitives. On the other hand, both sets of protocols utilize the MSG and LD/ST architectures in similar fashion.

Impact of load balancing framework. Comparing performance of BASE+DL and BASE+DL+LB versions in Figure 7.6, we find that not using the load-balancing framework decreases the achieved speedup on 64 nodes of the T3D to 5.4, 15.3, and 15.4 on the MSG, PUT/GET, and LD/ST architectures. The corresponding speedups on the Origin are limited to 0.8, 2.4, and 2.6 on 16 nodes for the three architectures.

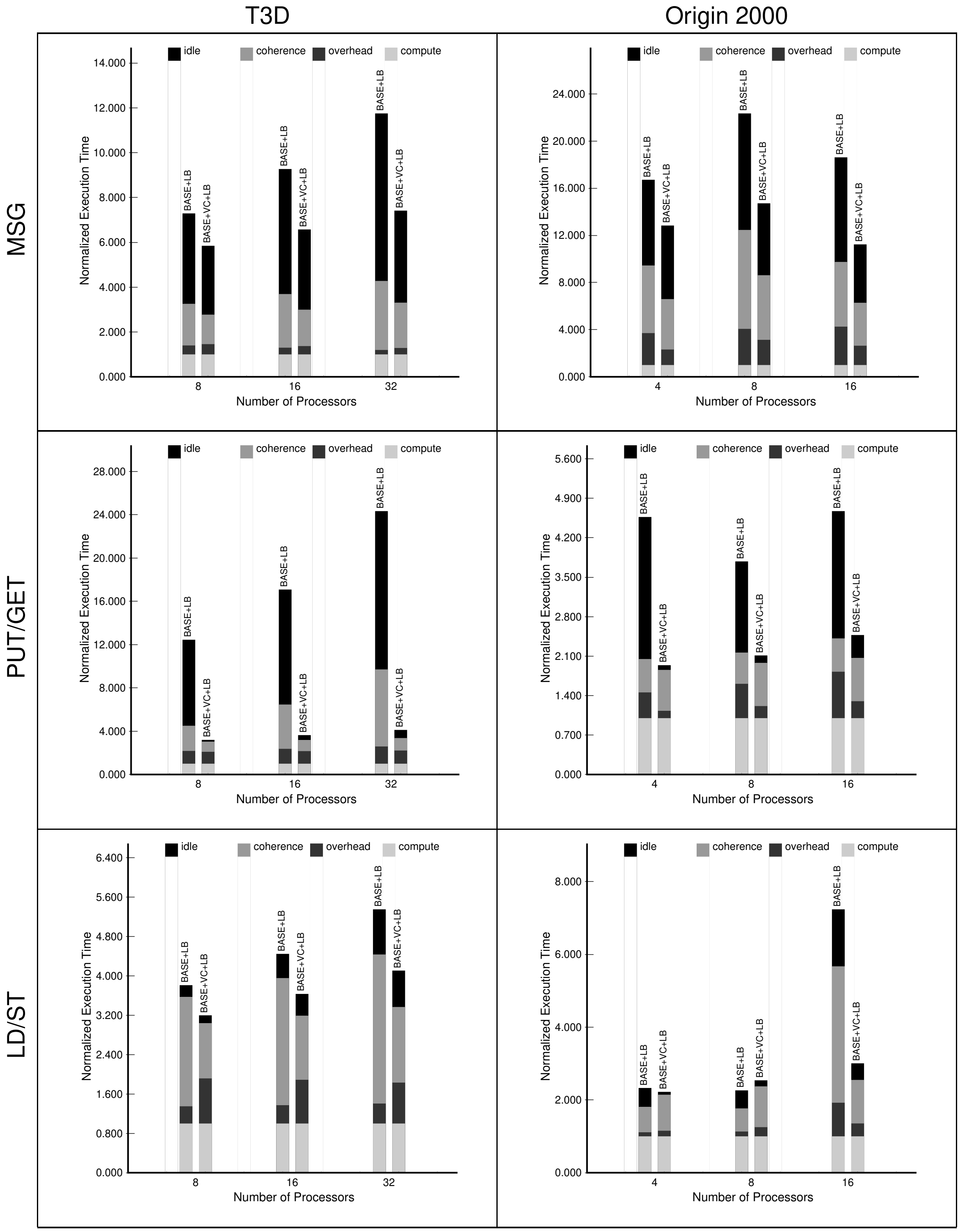

Figure 7.7 Execution time breakdown showing impact of data locality techniques for the IC-Cedar application.

To explain this loss in performance, we show a breakdown of the execution times for the two versions on a subset of processors in Figure 7.8. The components of the execution time are as described earlier. In the figure, the label BASE+DL+DATA denotes the use of an object placement mechanism, `GroupDistribution` for load balancing.

The data in the figure shows that using the load-balancing framework reduces both idle time and coherence overheads. Using a locality-aware load-balancing policy that relies on a spatial decomposition of the atom object collection results in fewer coherence actions, since most of the accessed objects are collocated. The decrease in idle time arises from the fact that a reduction in the number of coherence actions correspondingly reduces the amount of time a processor spends awaiting their completion. The figure shows that the performance improvement is most pronounced for the MSG architecture. On the other two architectures, lower overheads incurred by the view caching protocols are able to compensate for the increase in coherence traffic, so incorporating a locality-aware load balancing policy results in little additional benefit.

7.3 Radiosity

The radiosity method computes the global illumination in a scene containing diffusely reflecting surfaces by expressing the radiosity of an elemental surface patch as a linear function of the radiosities of all other patches, weighted by the distance and occlusion between the patches. Our application uses a sequential algorithm due to Hanrahan et al. [47], modeled after hierarchical N-body methods. We describe, in turn, the application structure, its use of view caching and load-balancing frameworks, and the resulting performance.

7.3.1 Application structure

The algorithm starts off with the initial patches comprising the scene and computes light transport between pairs of patches, hierarchically subdividing each patch as needed to ensure accuracy. This on-demand division of patches is responsible for reducing the complexity of the sequential algorithm from $O(N^2)$ to $O(N)$. However, the complexity reduction comes at a cost: the algorithm involves more sophisticated data structures (e.g., a quad-tree to maintain the subdivision structure), and more complex control structures (e.g., recursion over the child patches and data-dependent iteration). Thus, it is challenging to express even on sequential platforms.

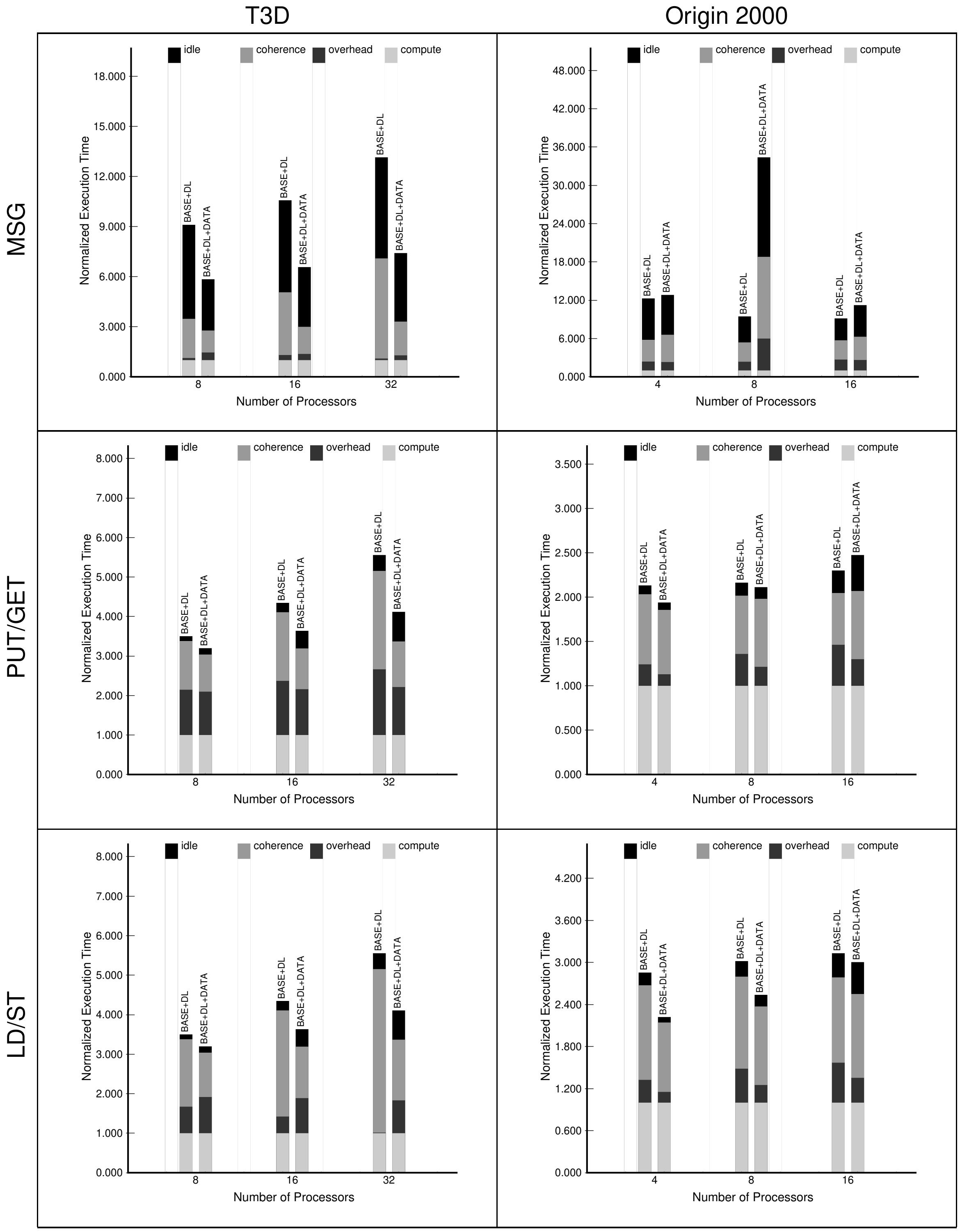

Figure 7.8 Execution time breakdown showing impact of load-balancing techniques for the IC-Cedar application.

The algorithm implementation uses two kinds of objects—patches and interactions—which are shown in Figure 7.9. Patches represent discrete elements of the diffusely reflecting surface, while interactions store information about the visibility and "formfactor"[1] of patch pairs. These objects are organized into three principal data structures: a quad-tree of patches, a list of interactions, and a binary space partitioning (BSP) tree of the input patches. Each patch maintains interaction lists of potentially visible neighbors (initially every other patch). Computation proceeds in iterations, where each patch's radiosity is computed as a linear combination of the radiosities of all patches on its interaction list, subdividing it or the other patches hierarchically as required to ensure accuracy (based upon an error computation). Radiosity computation requires that the visibility and formfactor of each interaction be previously calculated. Formfactor calculation depends entirely on the patch geometries, whereas the visibility calculation requires ray tracing. The visibility calculation dominates the computation time, and is efficiently implemented using the BSP tree. Subdivided patches are organized as a quad-tree, acquire their own interaction lists, and contribute a weighted term to their parent's radiosity. The iterations terminate when the change in the total radiosity (weighted sum of all input patches) falls below a threshold.

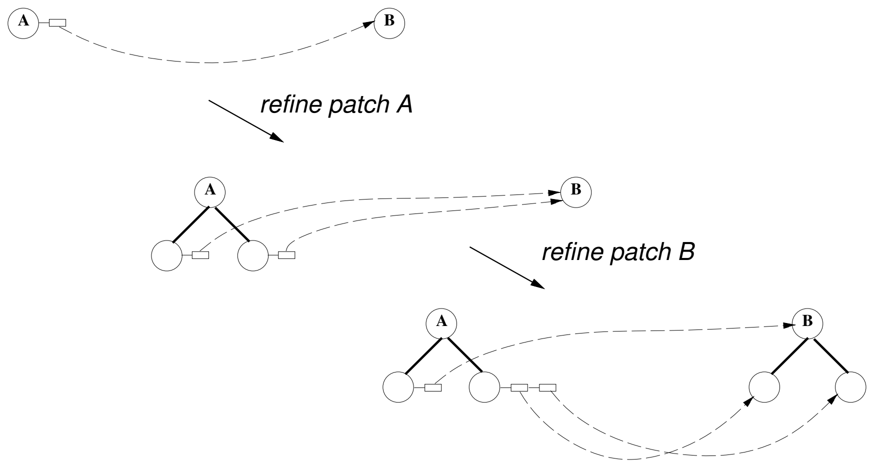

Figure 7.9 Patches (circles) and interactions (boxes) in the hierarchical radiosity application. Patches get refined on demand as the computation progresses.

[1] The formfactor determines the contribution of a patch's radiosity to other patches and depends upon the shape, distance, and relative orientation of the two patches.

Our parallel algorithm is derived from the radiosity application in the SPLASH-2 suite [168]. There are three levels of parallelism in each iteration: across all input patches (radiosity calculation), across child patches of a subdivided patch (radiosity calculation), and across neighbor patches stored in the interaction list (visibility, error, and radiosity calculation). Each of these tasks (radiosity calculation, visibility calculation, and error calculation). Both the thread and object structure exhibit data-dependent irregularity: in how deep a patch can be refined, in how many interactions need to be computed for each patch, and even in how long a particular interaction calculation takes. Additionally, work across the iterations is not uniform with $> 90\%$ of the computation occurring in the first iteration.

Thus, the radiosity application is characterized by dynamic threads created throughout the duration of execution. Although the thread creation structure is closely related to the data objects (a radiosity thread is created for each patch, and visibility and error threads are created for each interaction), the object structure evolves in a data-dependent fashion due to patch refinement. This results in an irregular and unpredictable thread structure. Additionally, threads exhibit wide granularity variations and access data objects in an unstructured fashion. Patch fields are updated every iteration, so they cannot be cached across iterations.

Performance of low-level approaches. Previous efforts [172] for efficiently executing the hierarchical radiosity application have primarily been restricted to hand-tuned implementations on cache-coherent shared memory machines. In fact, the irregular thread and data access structure of the application have often been used to motivate the need for such machines. Such implementations explicitly manage the creation and distribution of radiosity, visibility, and error calculation tasks, relying on the underlying hardware support to coherently access the patch objects. Previously published speedup numbers (obtained from [172] and [50]) on the same data set show that the application achieves a speedup of 26 on 32 processors of the DASH [128] and a speedup of 11 on 16 processors of the Origin.

7.3.2 Data locality and load-balancing optimizations

In contrast to the approach mentioned above, our focus is on achieving good performance on a variety of communication architectures without altering the dynamic multithreaded expression of the program. The primary performance challenges are data locality (for access to the patch and interaction objects) and load balance (for the radiosity, visibility, and error calculation threads). The BASE version of the application ensures data locality using a combination of data alignment and on-demand object caching: The interaction objects are aligned with the source patch, the BSP tree is explicitly replicated on each processor, and patch objects are cached using an object-consistent DSM. Load balance is achieved using the default mechanism of controlling object placement and executing threads local to their target objects. The input patches are uniformly distributed across the machine, but patches created due to refinement are allocated locally to preserve data locality. While the BASE version is relatively efficient for small numbers of processors, the irregular and data-dependent thread and object structure in Radiosity prevent it from achieving good performance on larger configurations. These shortcomings are alleviated using the view caching and load-balancing frameworks as summarized in Table 7.4 and described below.

Table 7.4 Optimizing radiosity performance using view caching and load balancing frameworks.

FRAMEWORK	WHERE APPLIED	POLICY CHOICES
view caching	read access to patch object by visibility thread to retrieve patch geometry	`READ_SYNC` exploits application semantics that access is to a disjoint subset of the patch object `READ_CONSTANT` exploits application semantics that access is to patch state that will remain unchanged
	read access to patch object by error calculation thread to retrieve patch radiosity	`READ_SYNC` exploits application semantics that access is to a disjoint subset of the patch object `READ_SELF_INV` exploits application semantics that patch state is required to be consistent only up to the last iteration
load balancing	allocation of refined patches	`ObjectPlacement` specifies random allocation to distribute radiosity and error calculation threads
	creation of visibility threads	`WorkSharing` `WorkSharingLocal` `WorkStealing` specifies thread placement policies for dynamic load balancing

Utilizing the view-caching framework. Coherence costs for caching remote patches are reduced by exploiting knowledge about application data access semantics. Two object views lend themselves to caching using custom protocols. The first is encountered in the visibility thread which accesses patch state (fields describing the patch geometry) that is known to be constant after initialization. This allows the view to be efficiently cached using either the `READ_SYNC` or the `READ_CONSTANT` protocols. The former eliminates false sharing and ensures that the cached view is never invalidated. The latter is more aggressive and obtains a detached view, relying on requester-initiated `access-grant` and `data-transfer`, and never requiring `access-revoke`.

The second object view is encountered in the error calculation thread which accesses patch state (the radiosity value) that is required to be consistent only up to the last iteration. This permits the view to be efficiently cached using either the `READ_SYNC` or the `READ_SELF_INV` protocols. The former eliminates false sharing exploiting the fact that the thread accesses a disjoint subset of patch state. The latter is more aggressive and exploits the application's ability to compute with inconsistent state. In this case, the custom protocol obtains a detached view, relying on requester-initiated `access-grant`, `data-transfer`, and `access-revoke`, and requiring home node participation only for data transfer. To ensure required consistency, all cached views are invalidated at the end of each iteration.

Augmenting the BASE version with these custom protocols produces the BASE+DL version of the application.

Utilizing the load-balancing framework. The dynamic nature of the application results in poor load balance for the radiosity and visibility calculation threads with the thread distribution strategy adopted in the BASE version. Our load balancing framework enables better load balance using a combination of object placement and thread placement mechanisms. Exploiting the fact that the thread creation structure of radiosity threads is closely related to the structure of the patch objects, we control thread distribution by controlling the placement of patch objects. We use the `ObjectPlacement` policy to create refined patches on random targets instead of locally, which results in a balanced distribution of the radiosity threads.

However, a similar approach does not suffice for the visibility threads. These threads require an explicit thread placement policy which is conveniently expressed at the creation site using our load balancing framework. All of the thread placement policies (`WorkSharing`, `WorkSharingLocal`, and `WorkStealing`) are applicable. Correlating the application structure with microkernel studies in Section 6.6, we find that no one policy is clearly superior, so detailed performance studies are required to determine which policy yields the best performance across the range of communication architectures.

Augmenting the BASE version with the above object and thread placement policies produces the BASE+LB version of the application.

7.3.3 Performance results

Figure 7.10 shows the best performance of the BASE, BASE+DL, BASE+LB, and BASE+DL+LB versions of the radiosity application for various communication architectures on the Cray T3D and the SGI Origin 2000.[2] As before, the Origin graphs show the performance of an additional version – BASE-CC – which corresponds to the BASE version executing on top of the CC-LD/ST interface to the hardware. The graphs show speedup with respect to the fastest single-node execution time (492.26 s on the T3D and 64.86 s on the Origin). These times are comparable to a sequential C++ version which executes in 379.56 s on the T3D and 58.77 s on the Origin. The larger difference on the T3D is attributable to inability of the generated code to effectively exploit the machine's fragile memory hierarchy.

The general trend seen in the graphs is that view caching and load balancing mechanisms significantly improve performance as compared to the BASE version across the entire range of communication architectures and target platforms. The following specific observations can be made about the collective and individual impact of the mechanisms.

Collective impact of mechanisms. On the T3D, the two mechanisms enable the BASE+DL+LB version to achieve peak speedups of 20.91, 34.69, and 33.71 on 64 nodes for the MSG, PUT/GET, and LD/ST architectures, respectively. The corresponding speedups on 32 nodes are 12.75, 21.47, and 20.94, which compare favorably with the previously reported speedup of 26 on 32 nodes of the DASH. This is particularly encouraging, given the architectural differences between the T3D and the DASH — the DASH has hardware support for cache-coherent shared memory and relatively faster communication (in terms of processor clocks), whereas the T3D requires an order of magnitude higher cost to access remote data.

A more direct comparison is possible on the Origin. The BASE+DL+LB version achieves peak speedups of 3.53, 7.68, and 8.69 on 16 nodes for the MSG, PUT/GET, and LD/ST architectures, respectively. These speedups are actually better than the BASE-CC version where additional locking and coherence overheads limit peak speedup to 5.32 on 16 nodes. Our performance is also comparable with

[2] The best performance for the BASE+DL+LB version is achieved using `READ_CONSTANT` and `READ_SELF_INV` view-caching protocols for data locality, and `ObjectPlacement` and `WorkSharing` policies for load balance. The BASE+DL and BASE+LB versions are derived from BASE+DL+LB as described in Section 7.1.

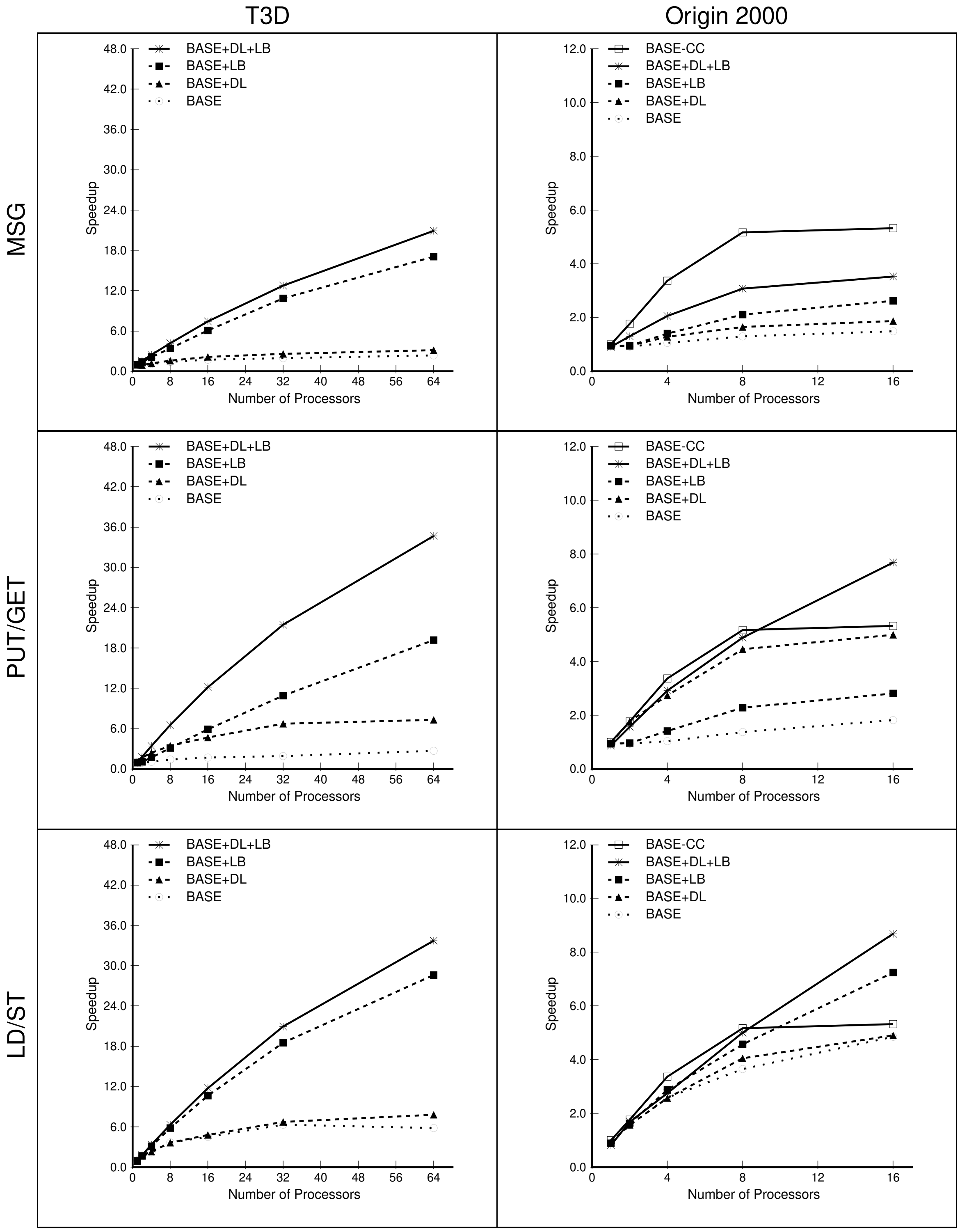

Figure 7.10 Overall performance impact of data locality and load-balancing techniques for the radiosity application.

the speedup of 11 on 16 nodes achieved by the SPLASH-2 version, which is promising because the latter includes significant tuning to reduce locking and coherence overheads.

Impact of view-caching framework. Comparing performance of BASE+LB and BASE+DL+LB versions, we find that not using the view-caching framework decreases the achieved speedup on 64 nodes of the T3D to 17.06, 19.18, and 28.60 for the MSG, PUT/GET, and LD/ST architectures, respectively. The corresponding speedups on the Origin are limited to 2.62, 2.81, and 7.24, respectively, on 16 nodes for the three architectures.

To explain this loss in performance, we examine the impact of view caching protocols in more detail. Figure 7.11 shows the performance of three application versions—BASE+LB, BASE+VC+LB, and BASE+NOFS+LB—for various communication architectures. The latter two versions represent different choices for the custom view caching protocols: BASE+VC+LB incorporates the `READ_CONSTANT` and `READ_SELF_INV` protocols, while BASE+NOFS+LB incorporates the `READ_SYNC` protocols (`NOFS` denotes the absence of false sharing). Figure 7.12 shows the execution time breakdown for these versions on a subset of processors. As before, the execution time consists of compute, overhead, coherence, and idle time components.

Figure 7.12 shows that the performance of view caching protocols stem from reduced coherence overheads and idle time. All of the `READ_SYNC`, `READ_CONSTANT`, and `READ_SELF_INV` protocols require reduced involvement of other processors: `READ_SYNC` eliminates false sharing while the latter two protocols require home node involvement only on the MSG architecture for data transfer. Consequently, the custom protocols incur reduced coherence overheads and idle time because they does not need to wait for remote processors to respond to coherence requests.

Comparing performance of BASE+VC+LB and BASE+NOFS+LB versions shows that the asynchronous and decoupled nature of `READ_CONSTANT`, and `READ_SELF_INV` protocols offer several performance benefits in addition to eliminating false sharing. In particular, on the T3D, the BASE+NOFS+LB version is limited to speedups of 18.64, 20.35, and 30.55 on 64 nodes for the MSG, PUT/GET, and LD/ST architectures, respectively. The corresponding speedups on 16 nodes of the Origin are 3.04, 3.20, and 5.56, respectively. This performance difference is most significant for the PUT/GET architecture. Unlike synchronous protocols both in object-consistent DSM and `READ_SYNC`, asynchronous protocols in `READ_CONSTANT` and `READ_SELF_INV` are able to take advantage of one-sided hardware communication primitives. On the other hand, all protocols utilize the MSG and LD/ST architectures in similar fashion.

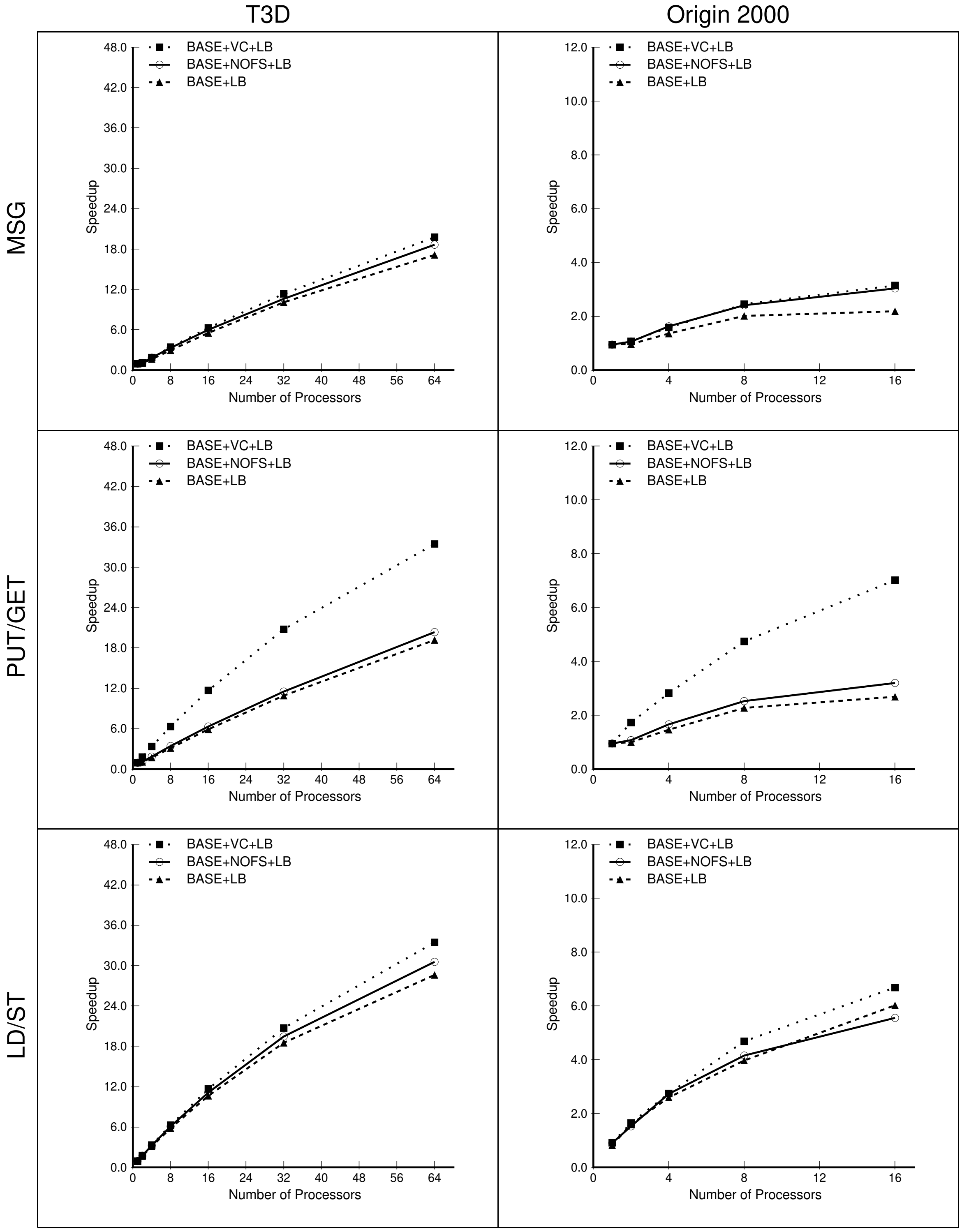

Figure 7.11 Individual impact of data locality techniques for the radiosity application.

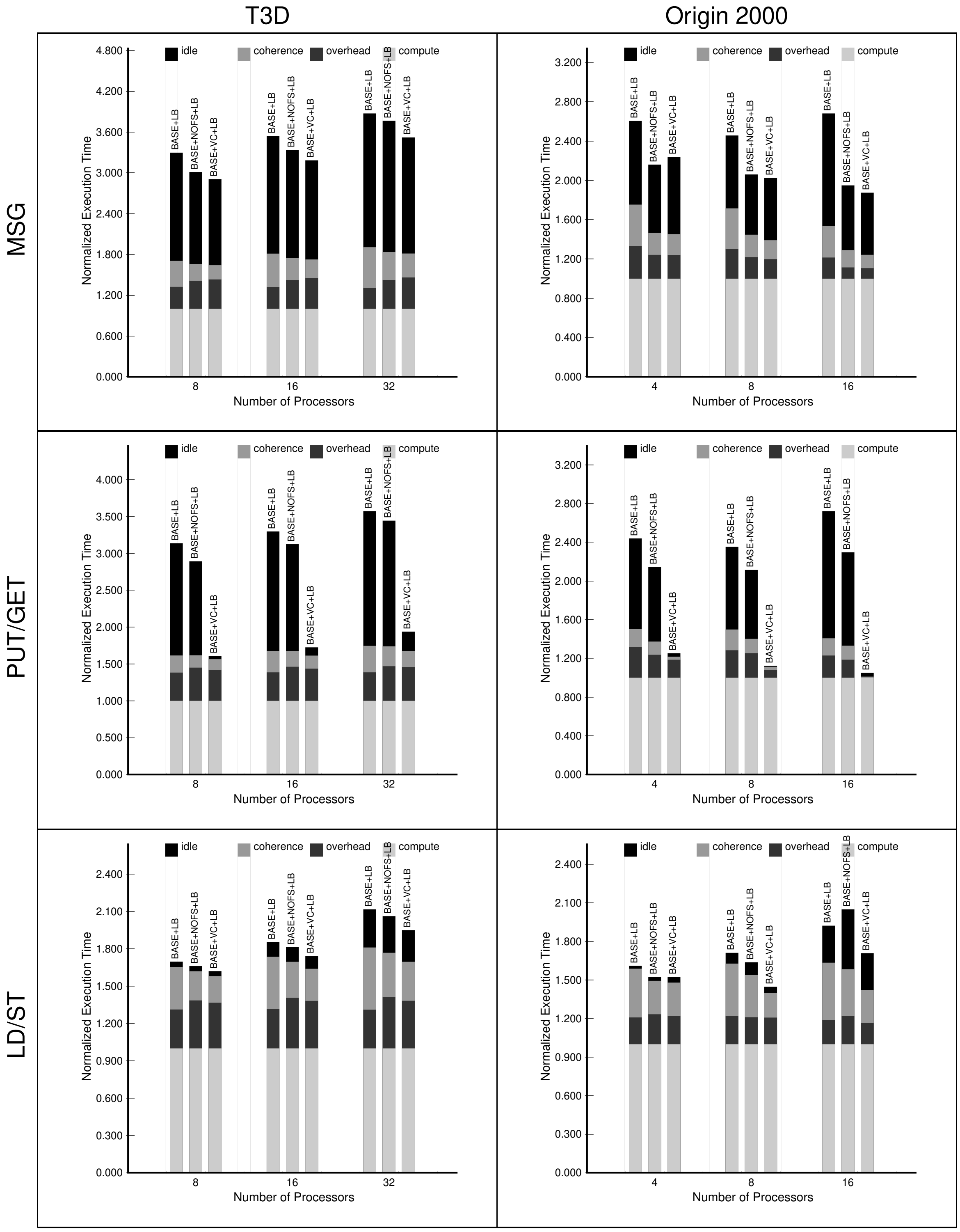

Figure 7.12 Execution time breakdown showing impact of data locality techniques for the radiosity application.

Impact of load-balancing framework. Comparing performance of BASE+DL and BASE+DL+LB versions in Figure 7.10, we find that not using the load-balancing framework decreases the achieved speedup on 64 nodes of the T3D to 3.14, 7.31, and 7.80 for the MSG, PUT/GET, and LD/ST architectures, respectively. The corresponding speedups on the Origin are limited to 1.87, 4.99, and 4.90, respectively, on 16 nodes for the three architectures.

To explain this loss in performance, we examine the impact of load-balancing mechanisms in more detail. Figure 7.13 shows the performance of five versions—BASE+DL, BASE+DL+DATA, BASE+DL+WSHR, BASE+DL+WSHRLOC, and BASE+DL+WSTL—for various communication architectures. The latter four versions represent different choices for load balancing: BASE+DL+DATA uses only object placement mechanisms while the other three versions use thread placement mechanisms. BASE+DL+DATA, BASE+DL+WSHR, BASE+DL+WSHRLOC, and BASE+DL+WSTL denote the `ObjectPlacement`, `WorkSharing`, `WorkSharingLocal`, and `WorkStealing` policies, respectively. Figure 7.14 shows the execution time breakdown for these versions on a subset of processors. As before, the normalized execution time consists of compute, overhead, coherence, and idle time components.

Figure 7.14 shows that using the load]-balancing framework primarily reduces idle time. The reduction in idle time comes from a balanced distribution of radiosity and visibility threads across the processors. The `ObjectPlacement` policy distributes patch objects so that radiosity calculations which execute local to them are uniformly distributed. All of the thread placement policies—`WorkSharing`, `WorkSharingLocal`, and `WorkStealing`—are approximately equally effective in improving load balance of visibility threads. The one exception is the LD/ST architecture where the `WorkStealing` policy demonstrates a slight performance advantage. These results substantiate trends observable in the context of microkernel studies described in Section 6.6.

The graphs in Figure 7.13 also show that object placement alone is insufficient to achieve high performance. Speedups for the BASE+DL+DATA version are limited to 10.56, 25.08, and 23.09 on 64 nodes of the T3D for the three communication architectures. Corresponding speedups on the Origin are limited to 2.95, 7.02, and 6.57 on 16 nodes.

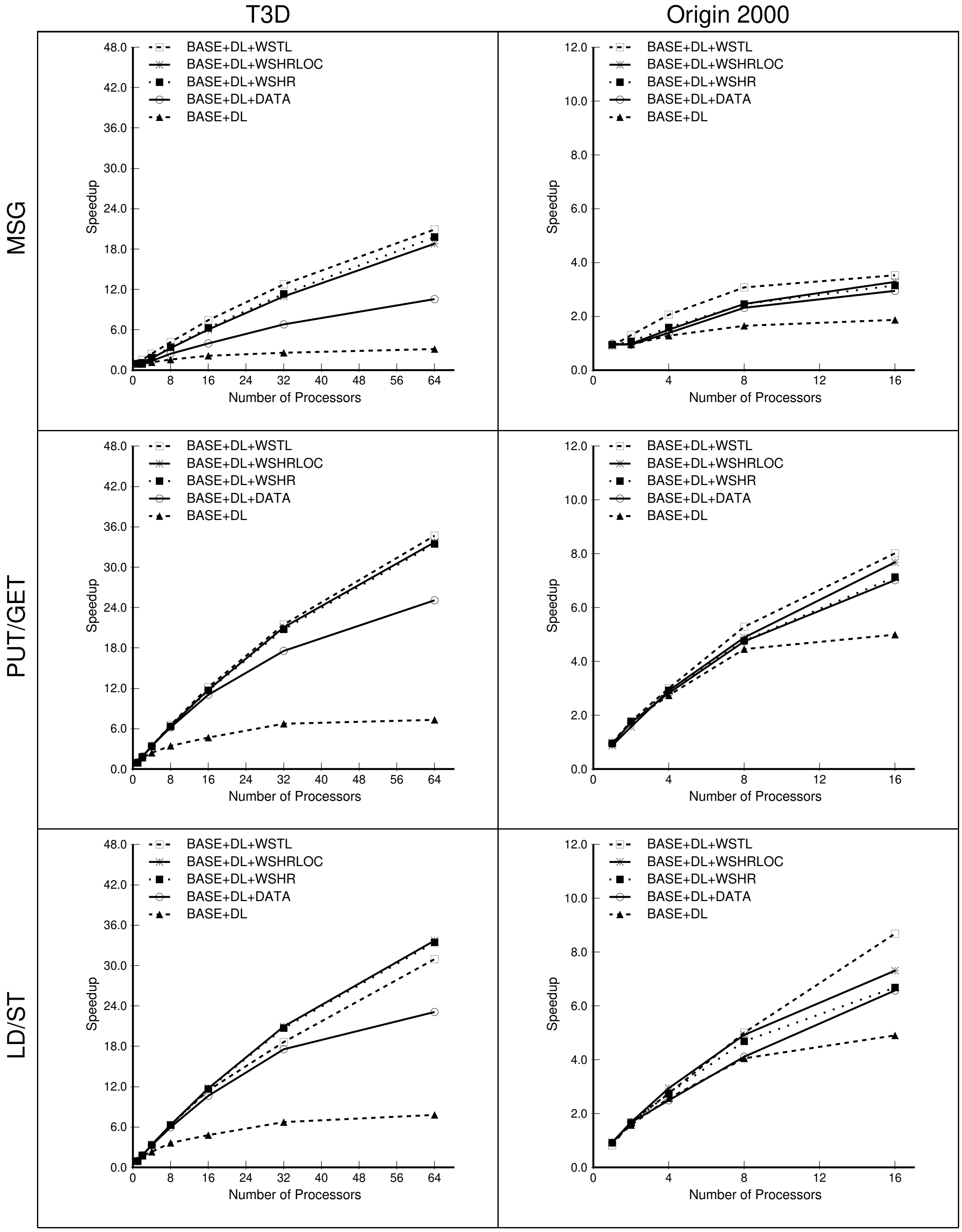

Figure 7.13 Individual impact of load-balancing techniques for the radiosity application.

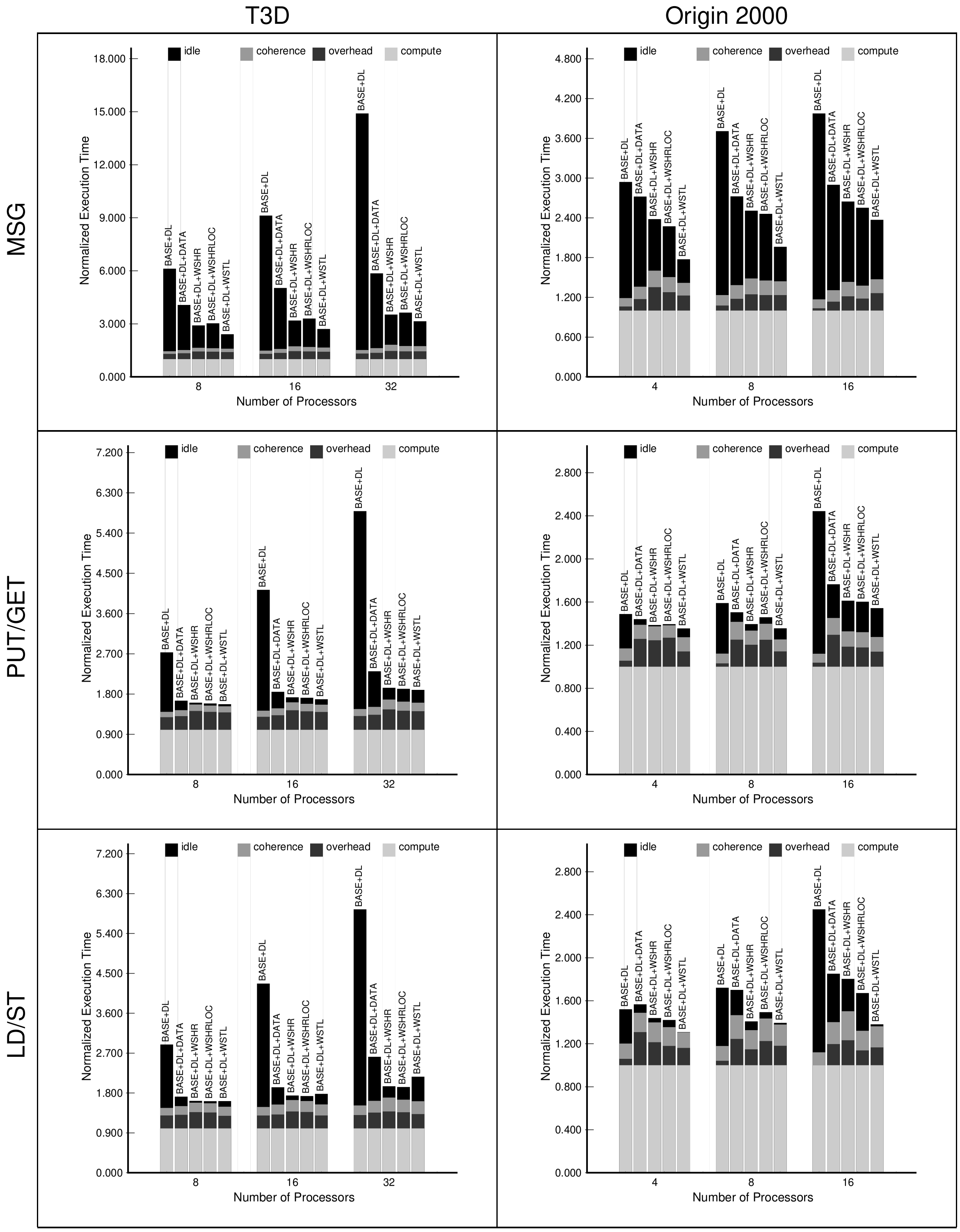

Figure 7.14 Execution time breakdown showing impact of load-balancing techniques for the radiosity application.

7.4 Gröbner

The Gröbner application computes the Gröbner basis of a set of polynomials. Our algorithm is based on a sequential algorithm due to Buchberger [178]. We describe, in turn, the application structure, its use of view caching and load-balancing frameworks, and the resulting performance.

7.4.1 Application structure

The algorithm starts off with an initial basis set of polynomials equal to the input set. Then, each possible pair of polynomials is evaluated, ranked according to a heuristic metric. Evaluation of a polynomial pair (which consists of arbitrary-precision arithmetic operations to isolate irreducible polynomial terms) can potentially lead to the addition of a new polynomial into the basis. This, in turn, generates additional polynomial pairs for evaluation which correspond to the new polynomial and existing basis polynomials. The algorithm completes when there are no polynomial pairs left for evaluation. The algorithm requires data structures that can represent arbitrary precision arithmetic on polynomials, as well as structures that can dynamically grow to store the basis. In addition, it requires iterative control structures with data-dependent termination criteria. Thus, the algorithm is challenging to express even on sequential platforms.

The algorithm implementation uses two objects—polynomials and the basis—shown in Figure 7.15. Polynomial objects store coefficients, exponents, and terms of polynomials being manipulated. The basis object implements a multiset abstraction and stores pointers to the different polynomials (using an array which expands as required). The parallelism in the application arises from parallel evaluation of polynomial pairs. Each pair creates a logical thread which accesses both the basis and polynomial objects. Threads which produce polynomials with irreducible terms invoke a method on the basis object to augment it and create new threads. The application exhibits data-dependent irregularity in how long each pair evaluation takes. Since the order of pair evaluations determine how the basis grows, they also affect the thread granularity.

Thus, the Gröbner application is characterized by threads created throughout the duration of execution. Although the thread structure is closely related to the data objects (threads are created for polynomial pairs), threads exhibit wide variations in granularity. Additionally, threads require prioritized execution to minimize redundant work. Threads concurrently read and write the basis object, so any cached copies must be kept coherent.

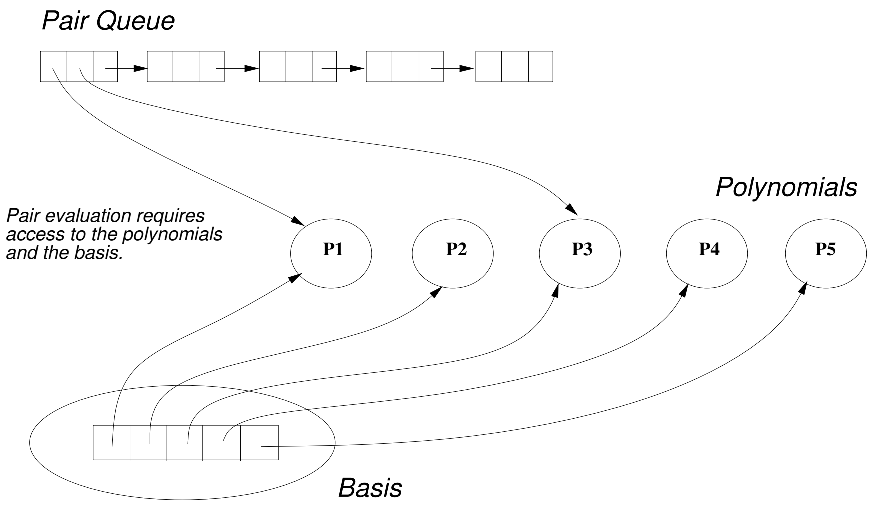

Figure 7.15 Polynomials, basis, and the pair queue in the Gröbner application. Pair evaluation adds new polynomials to the basis as the computation progresses.

Performance of low-level approaches. Previous efforts [169, 179] to efficiently execute the Gröbner application have either been restricted to small scale shared memory implementations [179] or have adopted a low-level approach with explicit locality management on distributed memory machines [169]. The latter approach explicitly replicates the polynomial and basis objects; all updates to the basis are explicitly propagated to its copies. Previously published speedup numbers (obtained from [169]) on similar data sets show that the application achieves a speedup of 6–10 times on 16 processors of the CM-5.[3]

7.4.2 Data locality and load-balancing optimizations

In contrast to the explicit replication approach described above, our focus is on achieving good performance without modifying the dynamic multithreaded expression of the program which assumes a global object space. The primary performance challenges are data locality (to access the polynomial and basis objects, the latter in the presence of concurrent updates) and load balance of the polynomial pair threads. The BASE version creates new polynomials on random targets, executes the pair evaluation

[3] The same source also reports speedup numbers for synthetically scaled benchmarks (each of which contain multiple copies of a smaller benchmark). However, we do not consider these results here because the synthetic benchmarks exhibit unconstrained parallelism not encountered in the original.

threads local to one of the objects, and accesses remote polynomials and the basis using an object-consistent DSM system. However, variations in thread granularity and thread imbalance prevent the BASE version from achieving good performance. Additionally, the BASE version exhibits severe contention for the shared basis object: A thread which is augmenting the basis object can prevent progress of other threads that need to read it. These shortcomings are alleviated using the view caching and load-balancing frameworks as summarized in Table 7.5 and described below.

Table 7.5 Optimizing Gröbner performance using view caching and load-balancing frameworks.

FRAMEWORK	WHERE APPLIED	POLICY CHOICES
view caching	read access to polynomial objects	`READ_CONSTANT` exploits application semantics that polynomial objects are not modified after creation
	read access to basis object	`READ_ASYNC` exploits application semantics that the basis is monotonically growing and initial pair evaluation can proceed with an out-of-date copy

load balancing	creation of pair evaluation threads	`WorkSharingPriority`
		`WorkStealingPriority`
		`WorkStealingGlobalPriority`
		specifies thread ordering policies which dynamically distribute threads while preserving thread priorities

Utilizing the view-caching framework. Coherence costs for caching remote polynomials and the basis object are reduced by exploiting knowledge about data access semantics. Two object views, one each for polynomials and basis objects, lend themselves to caching using custom protocols. The first view is encountered in the pair evaluation thread for accessing polynomial objects which are known to remain unchanged after initialization. This permits the view to be efficiently cached using the `READ_CONSTANT` protocol, which relies on requester-initiated `access-grant` and `data-transfer` and never requires `access-revoke`.

The second object view is encountered during read accesses to the basis object. Application semantics permit computation to proceed with out-of-date basis copies. The monotonic growth of the basis implies that a polynomial pair that is reducible by a subset of the basis is also reducible by the entire basis. These semantics permit the object view to be cached using the custom `READ_ASYNC` protocol, which relies on requester-initiated `access-grant` and `data-transfer`, and asynchronous `access-revoke`. Asynchronous invalidations are sufficient to ensure that the cached copies of the basis object do not become overly stale. In addition to a reduction in coherence overheads, the `READ_ASYNC` protocol also enables the overlap of pair evaluation threads with basis augmentations, potentially reducing idle time due to concurrency control.

Incorporating these custom protocols into the BASE version produces the BASE+DL version.

Utilizing the load-balancing framework. Wide granularity variations and priority sensitivity result in poor load balance for the pair evaluation threads using the distribution strategy adopted in the BASE version. Our load-balancing framework enables better load balance by permitting the distribution of pair evaluation threads to be controlled by a priority-sensitive dynamic load-balancing policy. Since thread priorities are unrelated to object structure, object-placement policies are not effective and yield essentially the same performance as the BASE version. All remaining policies—`WorkSharingPriority`, `WorkStealingPriority`, and `WorkStealingGlobalPriority`—are applicable. Our examination of application structure indicates that relatively few high-priority threads are created, suggesting that the `WorkStealingGlobalPriority` policy would yield the best performance. This hypothesis is supported by the performance results reported later in this section.

Augmenting the BASE version with these thread ordering policies produces the BASE+LB version of the application.

7.4.3 Performance results

Figure 7.16 shows the best performance of the BASE, BASE+DL, BASE+LB, and BASE+DL+LB versions of the Gröbner application for various communication architectures on the Cray T3D and the SGI Origin 2000.[4] As before, the Origin graphs show the performance of an additional version—BASE-CC— which corresponds to the BASE version executing on top of the CC-LD/ST interface to the hardware. The graphs show speedup[5] with respect to the fastest single-node execution time (486.87 s on the T3D and 72.08 s on the Origin). These times are comparable to a sequential C++ version which executes in 486.98 s on the T3D and (69.41 s on the Origin.

[4] The best performance for the BASE+DL+LB version is achieved using the `WorkStealingGlobalPriority` policy for load balance. The BASE+DL and BASE+LB versions are derived from BASE+DL+LB as described in Section 7.1.

[5] To prevent anomalous speedups, we ensure that new polynomials are added to the basis in the same order across all the versions.

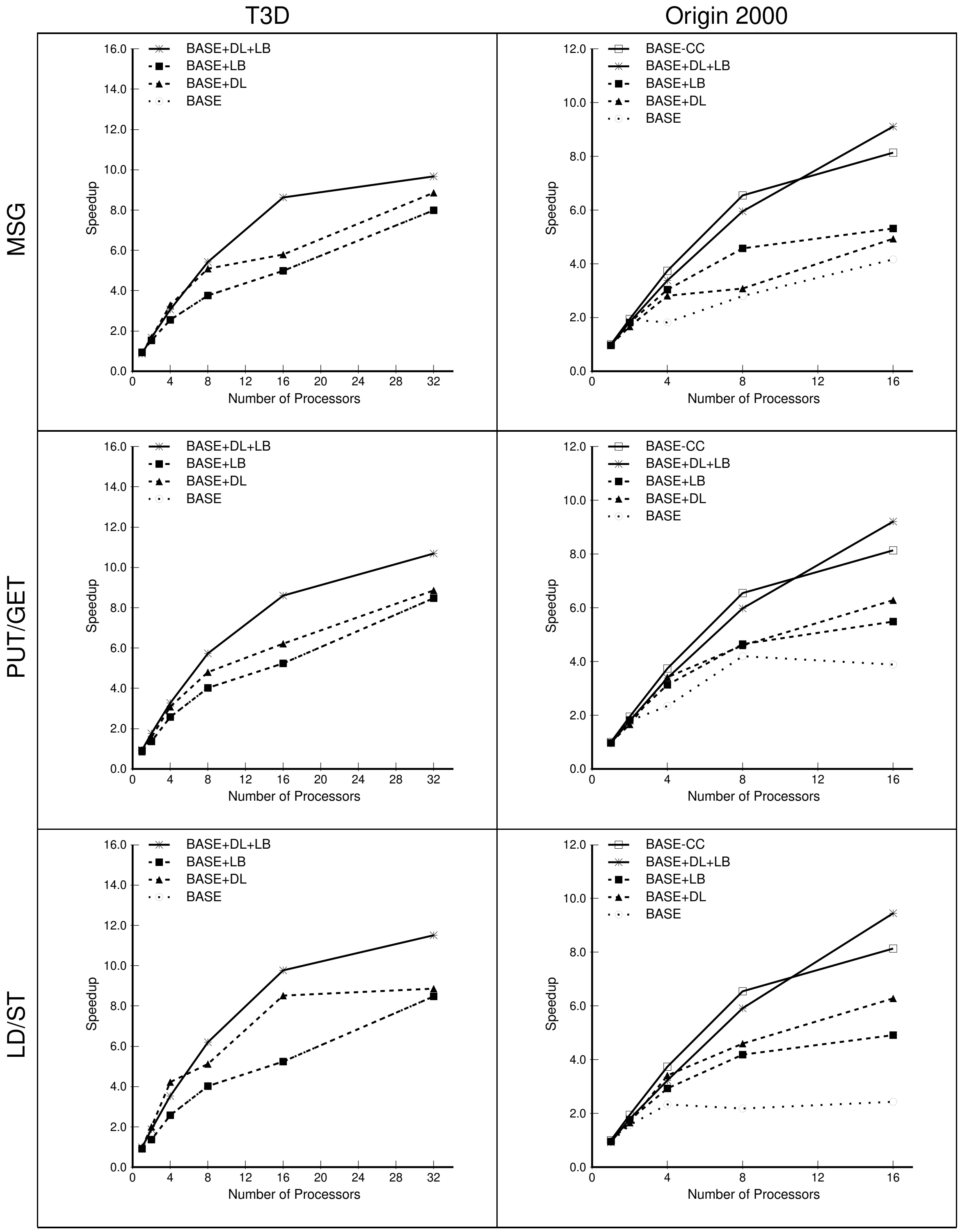

Figure 7.16 Overall performance impact of data locality and load balancing techniques for the Gröbner application.

The general trend seen in the graphs is that view-caching and load-balancing mechanisms significantly improve performance as compared to the BASE version across the entire range of communication architectures and target platforms. The following specific observations can be made about the collective and individual impact of the mechanisms.

Collective impact of mechanisms. On the T3D, the two mechanisms enable the BASE+DL+LB version to achieve peak speedups of 9.67, 10.69, and 11.51 on 32 nodes for the MSG, PUT/GET, and LD/ST architectures respectively. The corresponding speedups on 16 nodes are 8.63, 8.60, and 9.77 which compare favorably with the reported speedup of 6–10 on 16 nodes of the CM-5. These results are particularly encouraging, given the architectural differences between the CM-5 and the T3D — the CM-5 exhibits a more robust memory hierarchy and relatively faster communication (in terms of processor clocks) than the T3D.

On the Origin, the BASE+DL+LB version achieves peak speedups of 9.11, 9.21, and 9.45 on 16 nodes for the MSG, PUT/GET, and LD/ST architectures, respectively. These speedups are actually better than the BASE-CC version where concurrency control overheads due to contention for the shared basis limit peak speedup to 8.14 on 16 nodes.

Impact of view caching framework. Comparing performance of BASE+LB and BASE+DL+LB versions, we find that not using the view-caching framework decreases the achieved speedup on 32 nodes of the T3D to 7.99, 8.47, and 8.47 for the MSG, PUT/GET, and LD/ST architectures, respectively. The corresponding speedups on the Origin are limited to 5.31, 5.48, and 4.91, respectively, on 16 nodes for the three architectures.

To explain this loss in performance, we show a breakdown of the execution times for the two versions on a subset of processors in Figure 7.17. As before, the execution time is decomposed into compute, overhead, coherence, and idle time components and normalized with respect to the minimum compute time. In the figure, the label BASE+VC+LB denotes the use of view-caching mechanisms for data locality.

The data in the figure shows that using the view caching mechanisms reduces both idle time and coherence overhead. However, the dominant reduction in execution time stems from a reduction in idle time. The reduction primarily arises from the fact that the `READ_ASYNC` protocol obtains a detached view of the basis, permitting overlap of pair evaluation threads with basis augmentations. This overlap reduces the time a processor would otherwise spend waiting to obtain a valid copy of the view. The reduction in idle time due to improved concurrency control as opposed to asynchronous protocols also

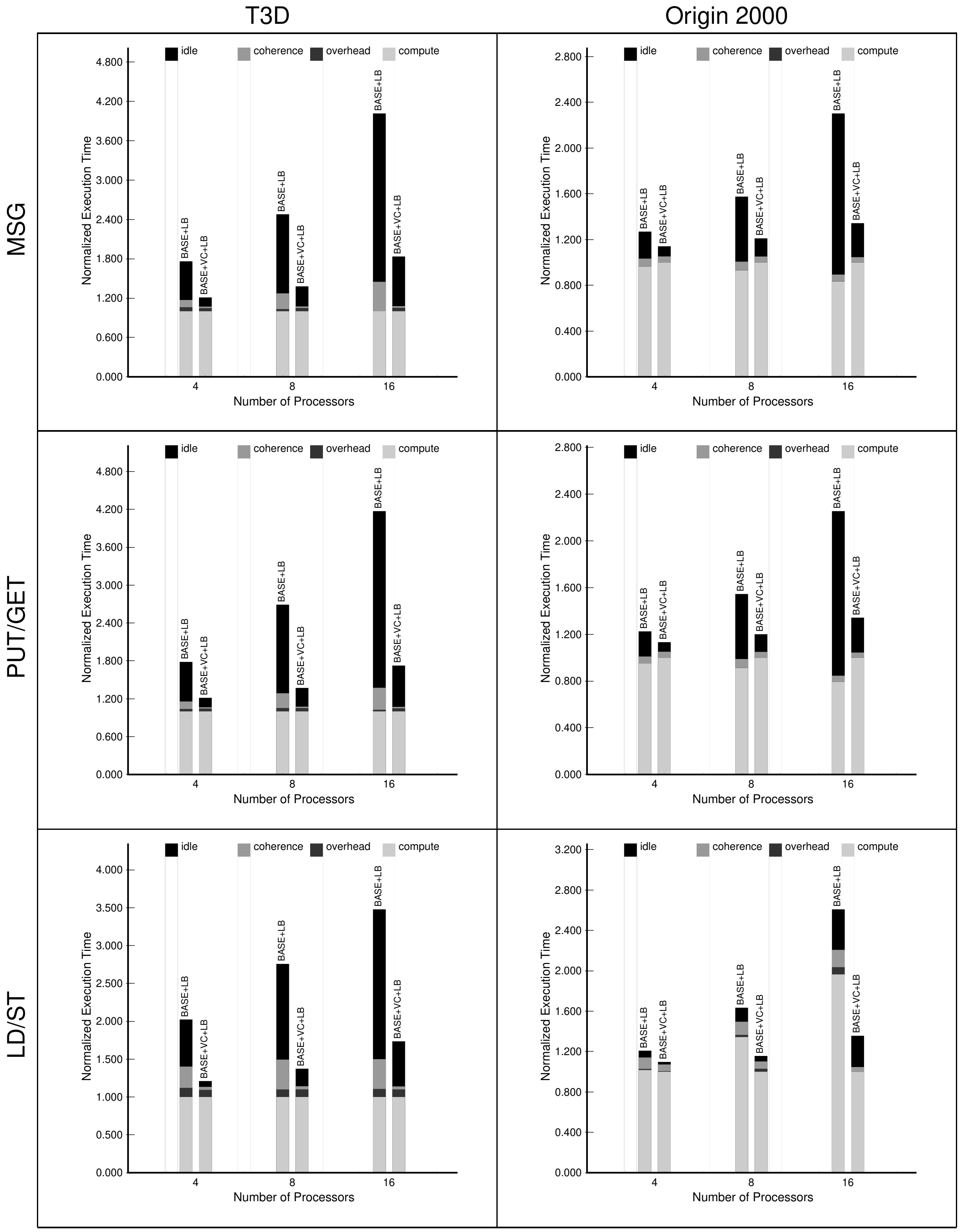

Figure 7.17 Execution time breakdown showing impact of data locality techniques for the Gröbner application.

shows up in the graphs in another way: Unlike the IC-Cedar and radiosity applications, performance of Gröbner is relatively insensitive to the underlying communication architecture.

Impact of load-balancing framework. Comparing performance of BASE+DL and BASE+DL+LB versions in Figure 7.16, we find that not using the load-balancing framework decreases the achieved speedup on 32 nodes of the T3D to 8.86, 8.86, and 8.86 for the MSG, PUT/GET, and LD/ST architectures, respectively. The corresponding speedups on the Origin are limited to 4.93, 6.28, and 6.28, respectively, for the three architectures.

To explain this loss in performance, we examine the impact of load balancing mechanisms in more detail. Figure 7.18 shows the performance of four versions—BASE+DL, BASE+DL+WSHR, BASE+DL+WSTL, BASE+DL+WSTLGBL—for various communication architectures. The latter three versions represent different choices for load balancing, denoting the `WorkSharingPriority`, `WorkStealingPriority`, and `WorkStealingGlobalPriority` respectively. Figure 7.19 shows the execution time breakdown for these versions on a subset of processors. As before, the normalized execution time consists of compute, overhead, coherence, and idle time components.

Figure 7.18 shows that the performance impact of the load-balancing framework is dependent on the specific thread-ordering policy used. Among the three policies, the best overall performance is achieved by `WorkStealingGlobalPriority`. The other two policies, `WorkSharingPriority` and `WorkStealingPriority` yield performance comparable to or slightly worse than the BASE version. Figure 7.19 provides an explanation for this behavior. The `WorkStealingGlobalPriority` policy incurs the least compute and idle times. The reduced compute time reflects the fact that this policy is most effective at preserving thread priorities and thus performs the minimum redundant work among all policies. The reduced idle time is a consequence of better load balance provided by a centralized queue. Our results show that for the Gröbner application, the priority preserving and load balancing advantages of the `WorkStealingGlobalPriority` policy outweigh any contention costs for the shared queue.

7.5 Phylogeny

The phylogeny application computes the evolutionary history of a set of species, representing the history as a tree with the path from the root to a species showing the evolutionary path of that species. Our algorithm is based on a sequential algorithm described by Jones [170]. We describe in turn the application structure, its use of view-caching and load-balancing frameworks, and the resulting performance.

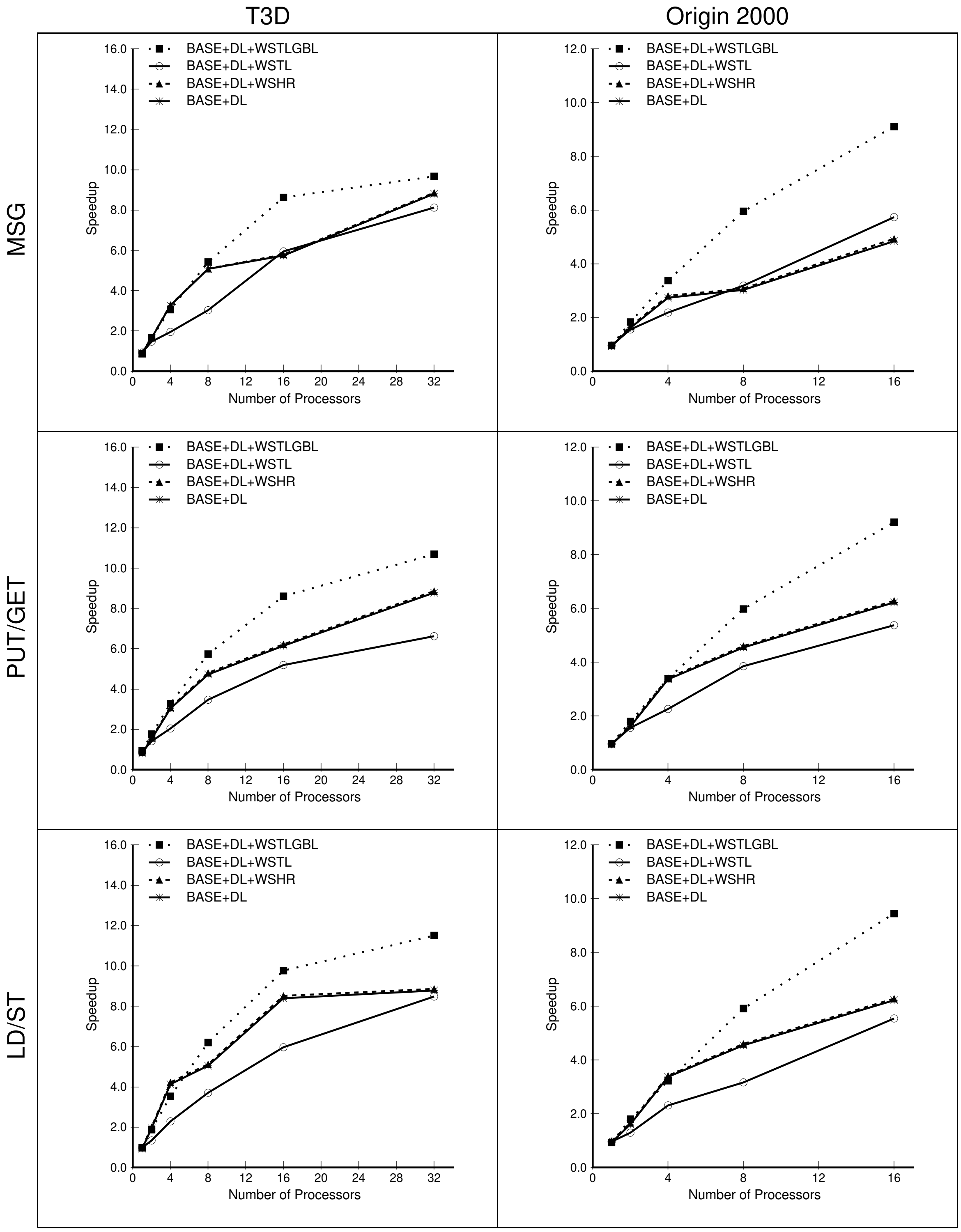

Figure 7.18 Individual performance impact of load balancing techniques for the Gröbner application.

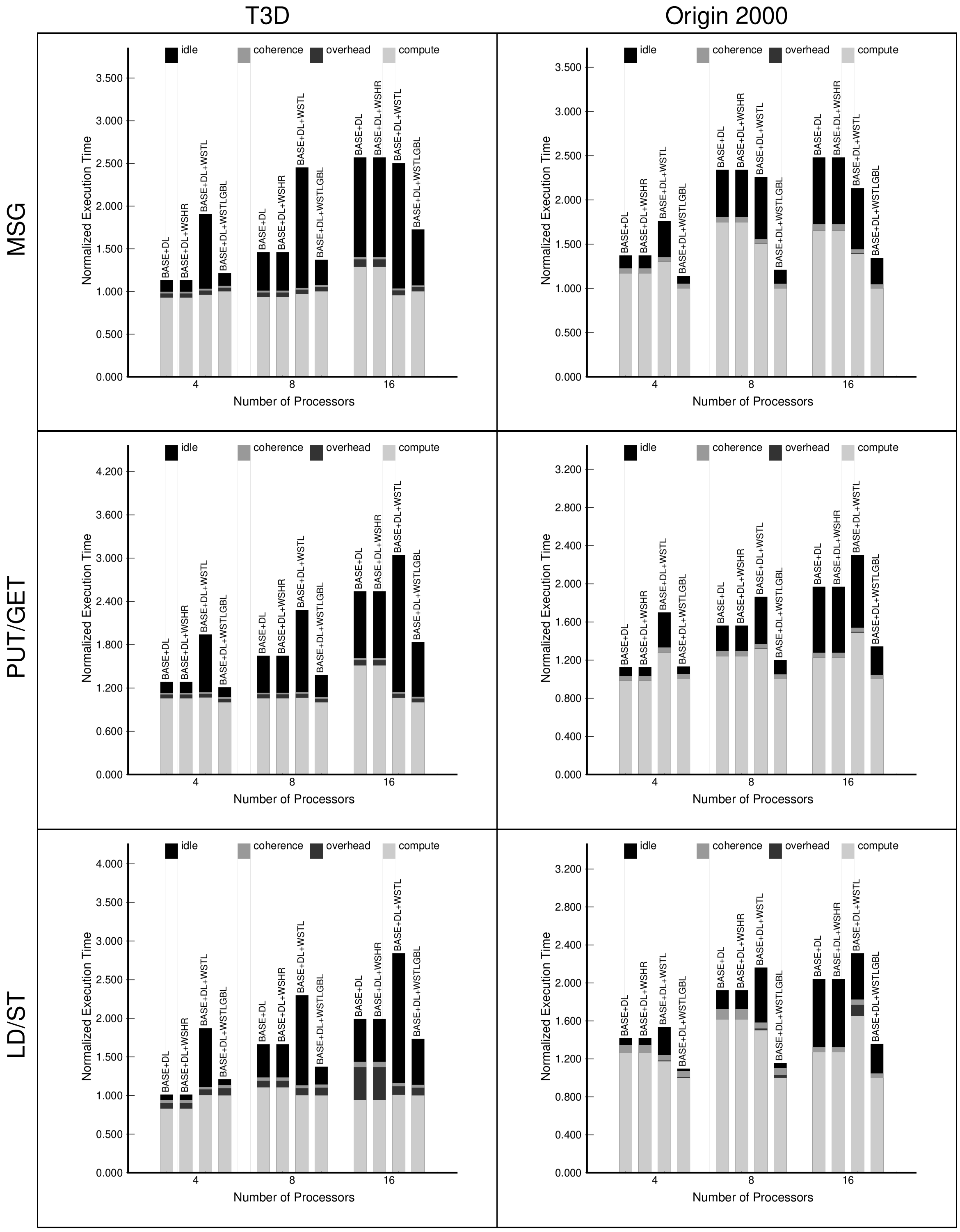

Figure 7.19 Execution time breakdown showing impact of load balancing techniques for the Gröbner application.

7.5.1 Application structure

Phylogenetic trees are constructed by considering the *characters* exhibited by the species. Given a set of species, with each species u represented as a vector of character values, $u[1], \ldots, u[c_{max}]$ (c_{max} is the maximum number of characters to be considered), a character is *compatible* with a phylogenetic tree if no value for that character arises more than once in any path in the tree. A *perfect phylogenetic tree* for a set of species and a given set of characters is a phylogenetic tree compatible with all the characters.

Our algorithm for the general phylogeny problem is based on a method known as *character compatibility* and uses a solution to the perfect phylogeny problem (determining whether or not a perfect phylogenetic tree exists) (see [170] for details) as a subroutine. The essence of the character compatibility method is a search for the largest compatible subset of characters, the rationale being that if the subset is large enough, the corresponding perfect phylogeny will be a good estimate of the evolutionary history of the species. The algorithm searches over the power set of characters (each possible subset) asking whether or not each subset is compatible. The algorithm maintains a failure set of subsets to facilitate pruning based upon the property that the subsets of a compatible set are also compatible. Computation corresponds to a bottom-up search of the character lattice (depth-first and right-to-left beginning with small subsets and progressing to larger subsets). Each subset is first checked against the failure set; no subset that has a subset that failed requires the perfect phylogeny procedure, otherwise the perfect phylogeny procedure is invoked. Based on the result of the procedure, either the subset is added to the failure set, or additional subsets are generated for exploration.

The implementation of the algorithm uses one kind of object—solution node — which is organized as a *trie* to implement the failure set (to facilitate efficient subset checking). Figure 7.20 pictorially shows tasks, solution nodes, and the failure set. Each solution node maintains a character subset, and child and parent pointers of the trie. Given the sophisticated nature of the data structures, and the pruning-based search procedure, the algorithm is complicated to express even on sequential platforms.

Our parallel algorithm follows the strategy of [170] and evaluates multiple character subsets in parallel. Given that adequate parallelism exists at this level, we have chosen to utilize a bundled solution to the perfect phylogeny problem as a (sequential) leaf subroutine. Each task traverses the global failure set which must be kept consistent against concurrent access, i.e., the failure set must be accessed and augmented within a critical section. The application exhibits data-dependent irregularity in how long each subset evaluation takes, both to check against the failure set and to determine a solution to the

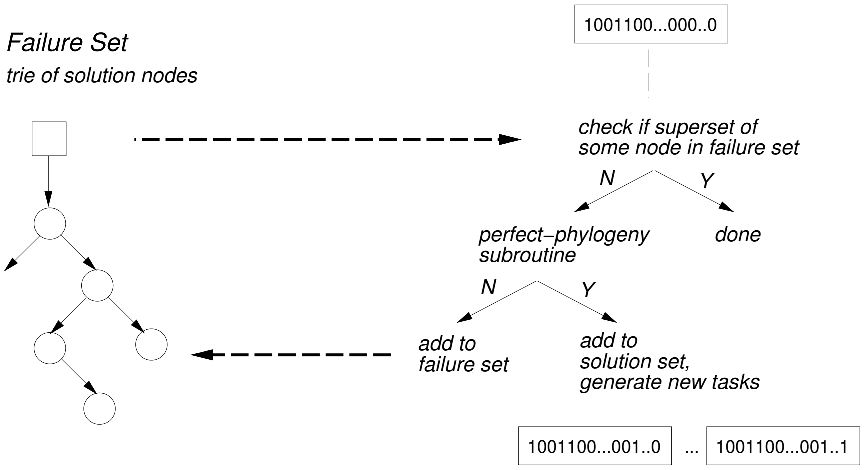

Figure 7.20 Tasks, solution nodes (circles), and the failure set (square) in the phylogeny application. Task evaluation adds new solution nodes into the failure set as computation progresses.

perfect phylogeny problem. Additionally, thread granularity varies with the order of subset evaluations (which affects how the failure set grows).

Thus, the phylogeny application is characterized by dynamic creation of threads spread throughout the duration of execution. These threads access the shared failure set trie in a data-dependent fashion and result in unstructured accesses to the component solution nodes. Threads exhibit imbalance in creation (only compatible subsets create additional threads) and wide variations in granularity. Additionally, threads require prioritized execution which corresponds to a depth-first and right-to-left traversal of the search space. Threads read and write the failure set which must be kept consistent against concurrent access.

Performance of low-level approaches. Previous efforts [170] to efficiently execute the phylogeny application have relied on a low-level approach which explicitly manages locality on distributed memory machines. The failure set is replicated across the processors with processors periodically synchronizing to make it consistent. Published speedup numbers (obtained from [170]) on similar data sets show that using this approach, the application achieves a speedup of 20 on 32 nodes of the CM-5.

7.5.2 Data locality and load-balancing optimizations

In contrast to the explicit replication approach described above, our focus is on achieving good performance without modifying the dynamic multithreaded expression of the program which assumes a global object space. The primary performance challenges are data locality (to access the solution node objects in the failure set), and load balancing the subset evaluation threads. The BASE version allocates new solution nodes on random target processors, executing threads local to a randomly selected solution node object. Local priority schedulers enforce thread priorities. Threads access remote objects using an object-consistent DSM system. However, variations in thread granularity, thread imbalance, and the need for prioritized execution prevent the BASE version from achieving good performance. Additionally, the BASE version exhibits severe contention for the failure set: A thread which is augmenting the set can prevent progress of other threads that need to read it.[6] These shortcomings are alleviated using the view caching and load-balancing frameworks as summarized in Table 7.6 and described below:

Table 7.6 Optimizing phylogeny performance using view caching and load balancing frameworks.

FRAMEWORK	WHERE APPLIED	POLICY CHOICES
view caching	read access to failure set for checking subset property	`READ_SELF_INV` exploits application semantics that subset evaluation can proceed with an inconsistent copy of the failure set
	write access to failure set for augmenting it with a new character subset	`WRITE_ALONE` exploits application semantics that there is at most one concurrently active writer
load balancing	creation of subset evaluation threads	`WorkSharingPriority` `WorkStealingPriority` `WorkStealingGlobalPriority` specifies thread ordering policies which dynamically distribute threads while preserving thread priorities

Utilizing the view-caching framework. Coherence costs for caching remote solution nodes are reduced by exploiting knowledge about data access semantics. Two object views, one for reading solution nodes and the other for writing them, lend themselves to caching using custom protocols.

The first view is encountered in the subset evaluation thread when it accesses the failure set to determine whether the current subset is a superset of some solution node in the set. The nature of the trie, and the fact that failing to find a subset in the failure set only results in an invocation of the perfect phylogeny routine, implies that task evaluation can proceed with an inconsistent copy of the failure set,

[6] This concurrency control is a consequence of the ICC++ locking protocol as implemented in the Concert compiler where read or write locks are obtained at the start of a method and released at the end. The program can be rewritten to introduce more concurrency by locking each solution node independently, but the associated overheads far outweigh the potential benefits.

i.e., inconsistent copies of the element solution nodes. This information about application semantics permits the use of the custom READ_SELF_INV protocol, which obtains a detached view, relying on requester-initiated access-grant, data-transfer, and access-revoke, and requiring home node participation only for data transfer. Cached copies of the view are self-invalidated whenever the subset evaluation thread needs to augment the failure set. In addition to a reduction in coherence overheads, the READ_SELF_INV protocol also enables the overlap of subset evaluation threads that read the failure set to check for supersets with the threads that augment it.

The second view is encountered in the subset evaluation thread when it augments the failure set. Since all augmentations are guarded by a lock on the root node of the failure set, there will be at most one active writer at a time. This information about application semantics permits the write view to be cached using the custom WRITE_ALONE protocol: updates are propagated back to the home node upon completion of the write.

Incorporating these custom coherence protocols into the BASE version produces the BASE+DL version of the application.

Utilizing the load-balancing framework. Granularity variations and priority sensitivity result in poor load balance for the subset evaluation threads using the distribution strategy adopted in the BASE version. Our load-balancing framework enables better load balance by permitting the distribution of subset evaluation threads to be controlled by a priority-sensitive dynamic load balancing policy. Thread priorities enforce the desired right-to-left, depth-first traversal of the character lattice. Since these priorities are unrelated to the solution node objects, load balancing mechanisms based on object placement are not effective, yielding essentially the same performance as the BASE version. All of the remaining thread ordering policies (WorkSharingPriority, WorkStealingPriority, and WorkStealingGlobalPriority) are applicable. Our examination of the application structure indicates that the computation is less sensitive to priority violations as compared to the Gröbner application, suggesting that the global priority ordering enforced by the WorkStealingGlobalPriority policy may not be required. This hypothesis is supported by the performance results reported later in this section.

Augmenting the BASE version with these thread-ordering policies produces the BASE+LB version of the application.

7.5.3 Performance results

Figure 7.21 shows the best performance of the BASE, BASE+DL, BASE+LB, and BASE+DL+LB versions of the phylogeny application for various communication architectures on the Cray T3D and the SGI Origin 2000.[7] As before, the Origin graphs show the performance of an additional version—BASE-CC—which corresponds to the BASE version executing on top of the CC-LD/ST interface to the hardware. The graphs show speedup with respect to the fastest single-node execution time (232.79 s on the T3D and 39.03 s on the Origin). These times are comparable to a sequential C++ version which executes in 200.97 s on the T3D and 47.22 s on the Origin. The larger execution time for the C++ version on the Origin is attributable to better memory management in the Concert run-time system.

The general trend seen in the graphs is that view caching and load balancing mechanisms significantly improve performance as compared to the BASE version across the entire range of communication architectures and target platforms. The following specific observations can be made about the collective and individual impact of the mechanisms.

Collective impact of mechanisms. On the T3D, the two mechanisms enable the BASE+DL+LB version to achieve peak speedups of 17.90, 23.52, and 26.61 on 32 nodes for the MSG, PUT/GET, and LD/ST architectures, respectively. These speedups compare favorably with the previously reported speedup of 20 on 32 nodes of the CM-5. Note that the performance is comparable despite architectural differences—the CM-5 has relatively faster communication (in terms of processor clocks)—and the low-level-programming approach, both of which favor higher speedups. Of course, the availability of richer communication architectures on the T3D further improve speedup.

On the Origin, the BASE+DL+LB version achieves speedups of 1.58, 2.08, and 2.04 on 16 nodes for the MSG, PUT/GET, and LD/ST architectures, respectively. While the performance is not very good, it is comparable to the BASE-CC version which only achieves a speedup of 3.44 on 16 nodes. This low performance of phylogeny is fundamental, arising from fine-grained reader-writer sharing at the level of solution nodes, which results in high overheads of coherence and synchronization operations (relative to user computation) even with hardware support for cache coherence. Similar behavior is not encountered on the T3D where the user computation incurs higher costs due to the large number of function calls. As expected, the MSG, PUT/GET, and LD/ST architectures show an additional loss in performance arising from use of software caching techniques.

[7] The best performance for the BASE+DL+LB version is achieved using the `WorkStealingPriority` policy for load balance. The BASE+DL and BASE+LB versions are derived from BASE+DL+LB as described in Section 7.1.

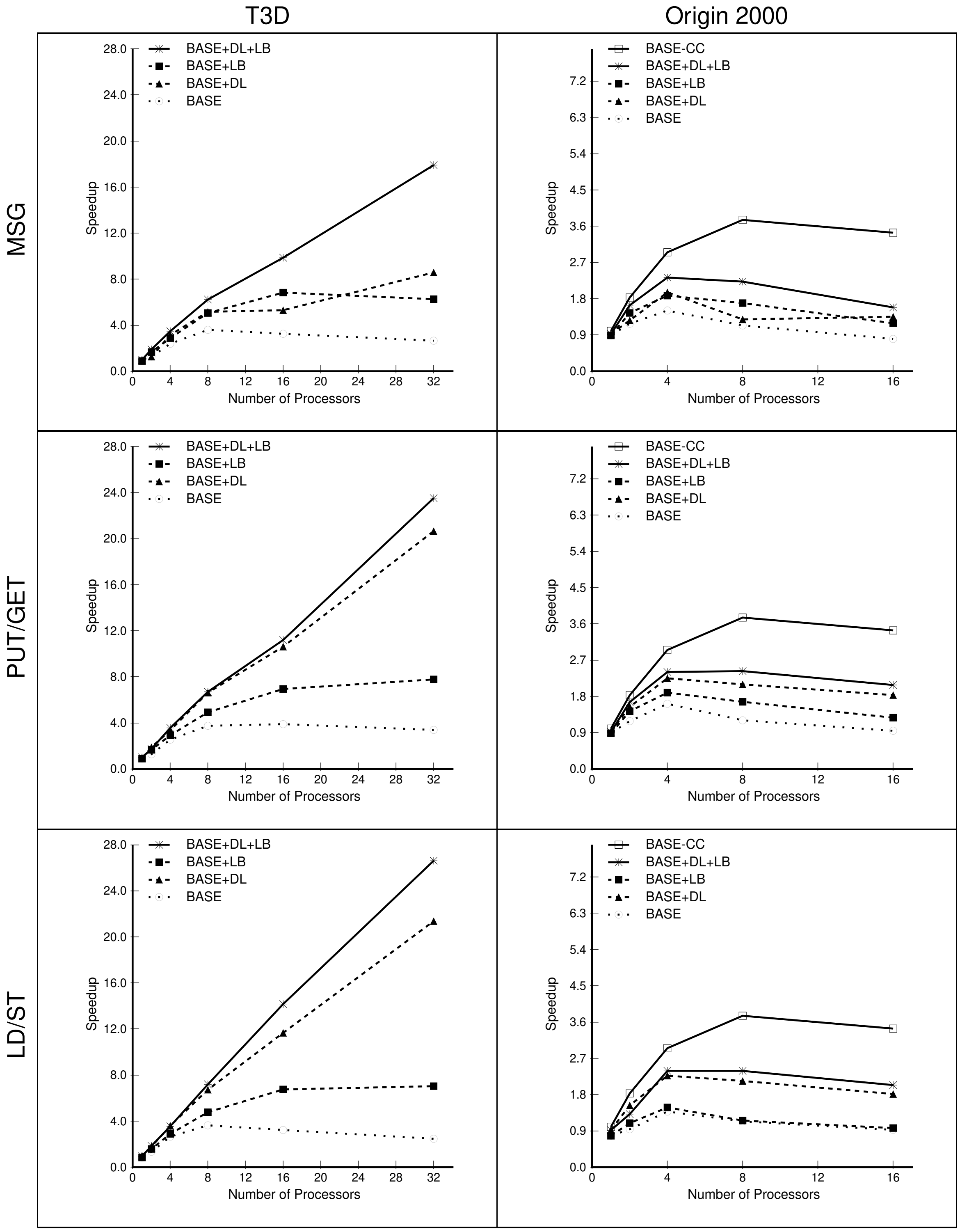

Figure 7.21 Overall performance impact of data locality and load-balancing techiques for the phylogeny application.

Impact of view-caching framework. Comparing performance of BASE+LB and BASE+DL+LB versions, we find that not using the view-caching framework decreases the achieved speedup on 32 nodes of the T3D to 6.26, 7.77, and 7.03 for the MSG, PUT/GET, and LD/ST architectures, respectively. The corresponding speedups on the Origin are limited to 1.19, 1.27, and 0.969, respectively, on 16 nodes for the three architectures.

To explain this loss in performance, we show a breakdown of the execution times for the two versions on a subset of processors in Figure 7.22. As before, the execution time is decomposed into compute, overhead, coherence, and idle time components and normalized with respect to the minimum compute time. In the figure, the label BASE+VC+LB denotes the use of view-caching mechanisms for data locality.

The data in the figure shows that using the view-caching mechanisms reduce idle time, coherence overheads, and compute time. However, the dominant reduction in execution time stems from a reduction in idle time. As in the Gröbner application, the reduction arises from improved concurrency control. The `READ_SELF_INV` protocol obtains a detached view of solution nodes, permitting overlap of threads which access the failure set to check for the superset property with augmentations to the failure set. This overlap reduces the time a processor would otherwise spend waiting to obtain a valid copy of the view.

Impact of load-balancing framework. Comparing performance of BASE+DL and BASE+DL+LB versions in Figure 7.21, we find that not using the load-balancing framework decreases the achieved speedup on 32 nodes of the T3D to 8.58, 20.65, and 21.36 for the MSG, PUT/GET, and LD/ST architectures, respectively. The corresponding speedups on the Origin are limited to 1.35, 1.83, and 1.81, respectively, on 16 nodes for the three architectures.

To explain this loss in performance, we examine the impact of load balancing mechanisms in more detail. Figure 7.23 shows the performance of four versions (BASE+DL, BASE+DL+WSHR, BASE+DL+WSTL, BASE+DL+WSTLGBL) for various communication architectures. The latter three versions represent different choices for load balancing, denoting the `WorkSharingPriority`, `WorkStealingPriority`, and `WorkStealingGlobalPriority` respectively. Figure 7.24 shows the execution time breakdown for these versions on a subset of processors. As before, the normalized execution time consists of compute, overhead, coherence, and idle time components.

Figure 7.23 shows that the performance impact of the load-balancing framework is dependent on the specific thread ordering policy which is used. Among the three policies, `WorkStealingPriority` yields the best overall performance, `WorkSharingPriority` yields performance comparable to the

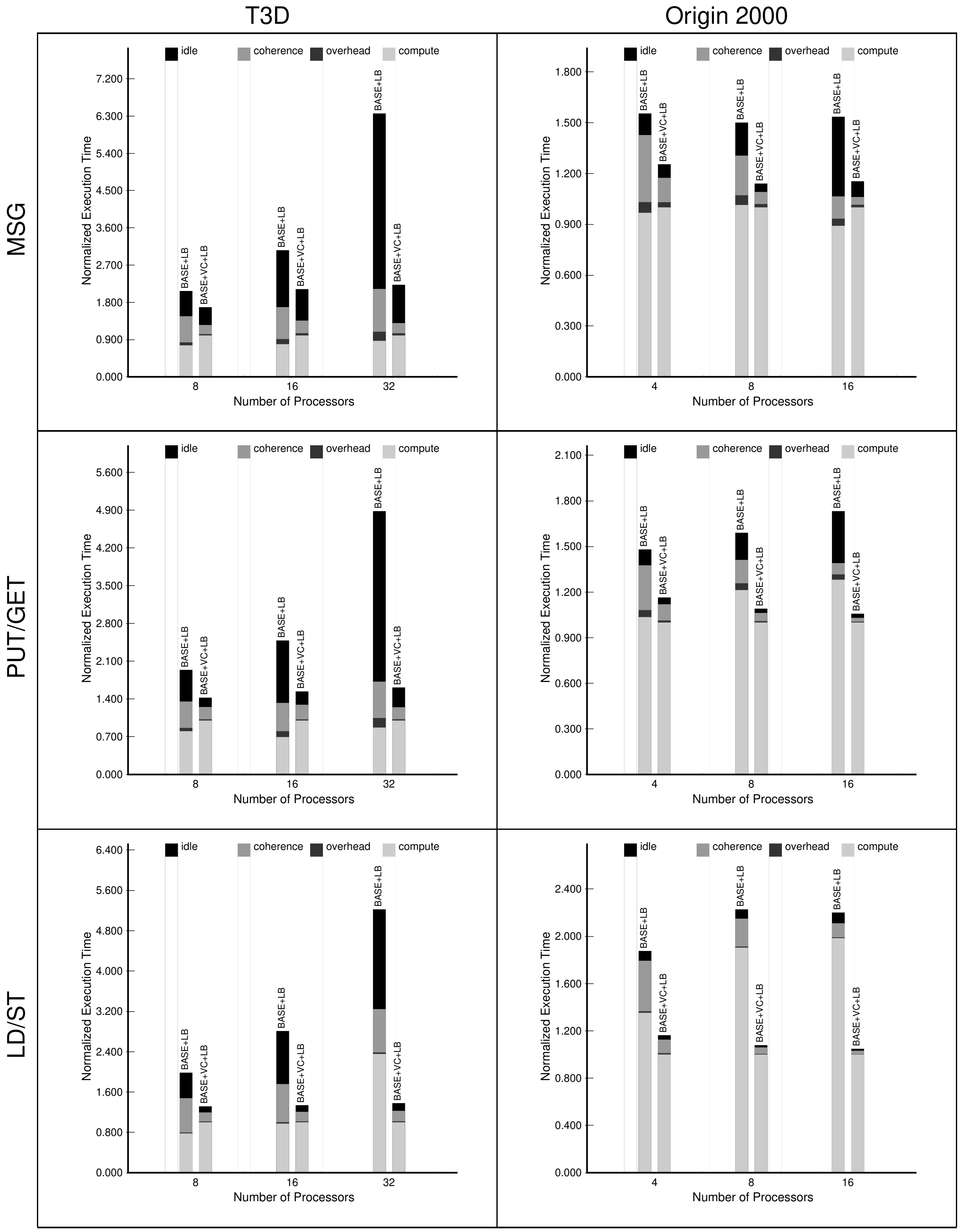

Figure 7.22 Execution time breakdown showing impact of data locality techniques for the phylogeny application.

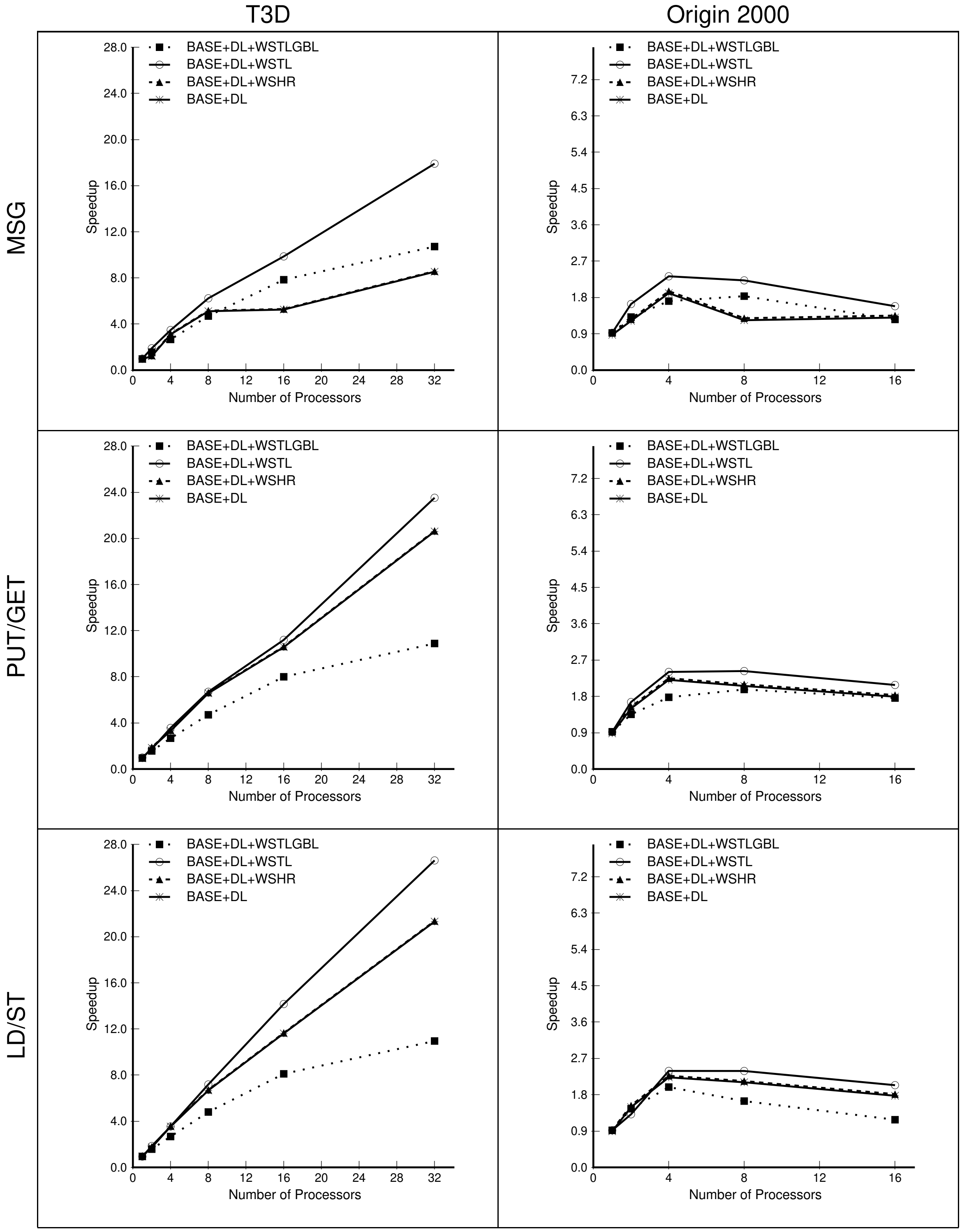

Figure 7.23 Individual performance impact of load balancing techniques for the phylogeny application.

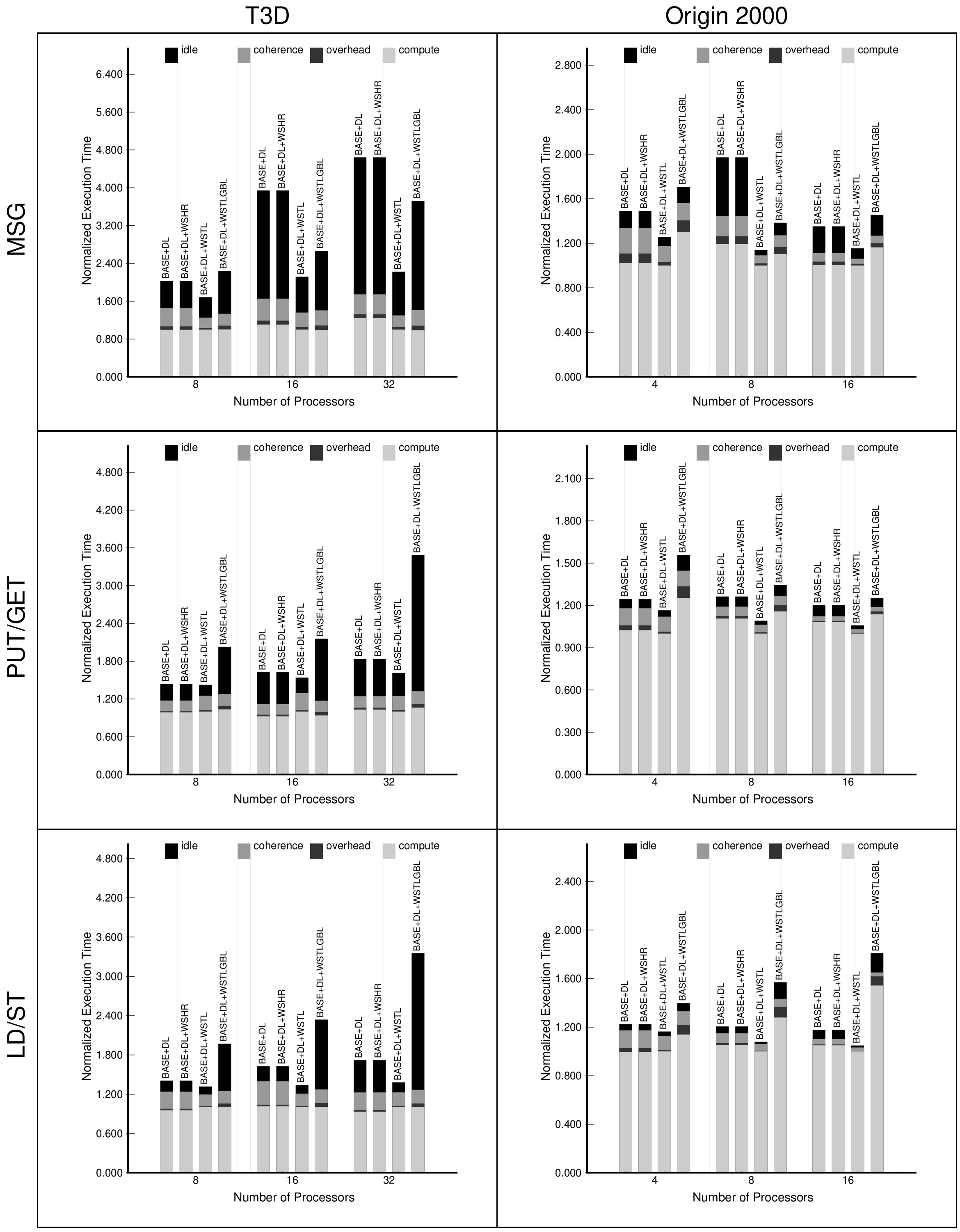

Figure 7.24 Execution time breakdown showing impact of load balancing techniques for the phylogeny application.

BASE version, and `WorkStealingGlobalPriority` yields the worst performance. Figure 7.24 provides an explanation for this behavior. The `WorkStealingPriority` policy incurs the least compute time, coherence overhead, and idle time. The compute time reflects the redundant work performed by the application when threads are executed out of priority order. To avoid redundant work, the phylogeny application requires that tasks be executed in an order which corresponds to a bottom-up traversal of the character lattice. The `WorkStealingPriority` policy preserves this order by keeping tasks on the processor where they were created. `WorkSharingPriority` destroys this order completely by distributing work upon creation. `WorkStealingGlobalPriority` maintains the priority order but unlike the Gröbner application, here the high contention costs (which show up as idle time) are not outweighed by its priority preserving advantages. In addition, the `WorkStealingPriority` policy also incurs reduced coherence overheads, and concomitant benefits in idle time, because it is better able to preserve locality among sibling tasks.

7.6 Overall Impact of Mechanisms

In this section, we analyze the use of view caching and load balancing frameworks across the application suite, and evaluate their performance impact. This section is organized as follows. Section 7.6.1 briefly describes the structure of each application. This structure is related in Section 7.6.2 to policy choices in the view caching and load balancing frameworks. Finally, Section 7.6.3 assesses the collective and individual impact of the two mechanisms.

7.6.1 Application structure

The thread and object structure of each of the four applications—IC-Cedar, radiosity, Gröbner, and phylogeny—is briefly reprised below:

- *IC-Cedar* is the least irregular of all the applications. All objects in the application are allocated at startup. Computation proceeds in iterations, with each iteration creating a fixed number of threads that are closely related to the object structure. Threads access objects in an unstructured fashion, based on spatial locality amongst objects. Object access patterns are not affected by ongoing computation. Shared objects are read and modified in different global phases in each iteration. Threads exhibit moderate variations in granularity.

- *Radiosity* exhibits irregularity in data structure, parallel control structure, and concurrency. Objects and threads are created in a data-dependent fashion throughout the duration of execution. Thread creation is unbalanced, exhibiting variation in the number of child threads. Threads access objects in an unstructured fashion which evolves with the computation. Objects are concurrently read and written. Threads exhibit wide variations in granularity.

- *Gröbner* is characterized by dynamic, data-dependent creation of objects and threads throughout the duration of execution. Thread creation is unbalanced. Threads access objects in a data-dependent unstructured fashion. High levels of contention are encountered for accessing a single object, the basis, which is concurrently read and modified. Threads exhibit wide variations in granularity and additionally require prioritized execution to minimize redundant work. Thread priorities are unevenly distributed with a relatively small number of high priority threads.

- *Phylogeny* is characterized by dynamic, data-dependent creation of objects and threads throughout the duration of execution. As with radiosity and Gröbner, thread creation is unbalanced. Threads access objects in a data-dependent unstructured fashion. Like Gröbner, a single object, the failure set, encounters high levels of contention and is concurrently read and modified. Threads exhibit wide variations in granularity and additionally require prioritized execution to minimize redundant work. Unlike Gröbner, phylogeny exhibits a larger number of high priority threads.

Thus, these applications exhibit irregularity in various dimensions such as thread structure, object access patterns, object sharing, etc. In the next section, we shall relate these application attributes to choice of view caching and load balancing policies for data locality and load balancing optimizations.

7.6.2 Data locality and load-balancing optimizations

We describe in turn the use of view caching and load-balancing frameworks across the application suite.

Utilizing the view-caching framework. The framework enhances data locality in the presence of unstructured, data-dependent access to objects which are concurrently read and modified. Table 7.7 summarizes the use of view-caching protocols for each of the applications.

The custom protocols improve performance over the synchronous object-consistent DSM protocols by requiring reduced remote processor involvement and synchronization. The `READ_CONSTANT`,

Table 7.7 Optimizing data locality using the view-caching framework. When there are multiple applicable policies, the policy that yields the best performance is distinguished with an asterisk.

APPLICATION	WHERE APPLIED	POLICY CHOICES
IC-Cedar	read access to atom objects to retrieve position fields	`READ_SELF_INV` exploits application semantics that position fields get updated every iteration

Radiosity	read access to patch object by visibility thread to retrieve patch geometry	`READ_SYNC`
		exploits application semantics that access is to a disjoint subset of the patch object
		`READ_CONSTANT*`
		exploits application semantics that access is to patch state that will remain unchanged

	read access to patch object by error calculation thread to retrieve patch radiosity	`READ_SYNC`
		exploits application semantics that access is to a disjoint subset of the patch object
		`READ_SELF_INV*`
		exploits application semantics that patch state is required to be consistent only up to the last iteration

Gröbner	read access to polynomial objects	`READ_CONSTANT` exploits application semantics that polynomial objects are not modified after creation
	read access to basis object	`READ_ASYNC` exploits application semantics that the basis is monotonically growing and initial pair evaluation can proceed with an out-of-date copy
Phylogeny	read access to failure set for checking subset property	`READ_SELF_INV` exploits application semantics that subset evaluation can proceed with an inconsistent copy of the failure set
	write access to failure set for augmenting it with a new character subset	`WRITE_ALONE` exploits application semantics that there is at most one concurrently active writer

READ_SELF_INV, and WRITE_ALONE protocols obtain a detached view which only requires home node involvement for data transfer. The READ_ASYNC protocol obtains an asynchronous view which relies on requester-initiated access-grant and data-transfer, and revokes access using an asynchronous protocol. Even the READ_SYNC protocol reduces processor involvement by eliminating false sharing. These custom protocols are enabled by four different kinds of knowledge about application data access semantics:

1. *Knowledge that different threads access disjoint subsets of the object state* enables use of the READ_SYNC protocol as in the Radiosity application.

2. *Knowledge that the accessed subset of object state is not going to be modified* enables use of the READ_CONSTANT protocol as in the Radiosity and Gröbner applications. Additionally, knowing that the accessed subset is not going to be modified in the current phase permits use of the READ_SELF_INV protocol as in the IC-Cedar application.

3. *Knowledge that computation can proceed with inconsistent values for the accessed subset of object state* enables use of READ_SELF_INV and READ_ASYNC protocols as in the Radiosity, Gröbner, and Phylogeny applications.

4. *Knowledge of concurrent activities on the object* enables use of the WRITE_ALONE protocol as in the Phylogeny application.

Utilizing the load-balancing framework. The framework enables balanced thread distribution in the presence of unbalanced thread creation, variations in thread granularity, and a need for prioritized execution. Table 7.8 summarizes the use of load-balancing policies for each of the applications.

The load-balancing policies improve performance over the default thread distribution strategy by enabling balanced thread distribution which is additionally sensitive to data locality and execution priorities. The use of the load-balancing framework across the application suite achieves three goals:

1. *Locality-sensitive thread distribution* as in the IC-Cedar, radiosity, and phylogeny applications which use the GroupDistribution, ObjectPlacement, and WorkStealingPriority policies respectively. The first two policies achieve thread distribution by controlling the placement of the target object and are suitable in applications where the thread creation structure is closely related to the object structure. The third policy preserves locality by grouping together threads which access the same set of data objects.

Table 7.8 Optimizing load balance using the load-balancing framework. When there are multiple applicable policies, the policies that yield the best performance are distinguished with an asterisk.

APPLICATION	WHERE APPLIED	POLICY CHOICES
IC-Cedar	allocation of atom object collection	`GroupDistribution` specifies a custom spatial decomposition based on Peano-Hilbert encoding
Radiosity	allocation of refined patches	`ObjectPlacement` specifies random allocation to distribute radiosity and error calculation threads
	creation of visibility threads	`WorkSharing`* `WorkSharingLocal`* `WorkStealing`* specifies thread placement policies for dynamic load balancing

Gröbner	creation of pair evaluation threads	`WorkSharingPriority`
		`WorkStealingPriority`
		`WorkStealingGlobalPriority`*
		specifies thread ordering policies which dynamically distribute threads while preserving thread priorities

Phylogeny	creation of subset evaluation threads	`WorkSharingPriority`
		`WorkStealingPriority`*
		`WorkStealingGlobalPriority`
		specifies thread ordering policies which dynamically distribute threads while preserving thread priorities

2. *Balanced thread distribution* as in the radiosity application which can use the `WorkSharing`, `WorkSharingLocal`, or `WorkStealing` policies. These policies are suitable for applications which exhibit unbalanced thread creation and wide variations in thread granularity. Choosing among the three policies is dependent on other application attributes such as locality sensitivity. In the specific case of radiosity, all policies yield approximately the same performance.

3. *Priority-sensitive thread distribution* as in the Gröbner and phylogeny applications which can use the `WorkSharingPriority`, `WorkStealingPriority`, or `WorkStealingGlobalPriority` policies. These policies are suitable for applications which exhibit unbalanced thread creation and require prioritized thread execution for efficiency. Selecting among them depends upon the criticality of prioritized execution, and other attributes such as locality sensitivity. In the specific cases

of Gröbner and phylogeny, the `WorkStealingGlobalPriority` and `WorkStealingPriority` policies, respectively, yield the best performance.

7.6.3 Performance impact

Collective impact. Figure 7.25 shows the aggregate impact of the view-caching and load-balancing mechanisms for each application on various communication architectures on the Cray T3D and the SGI Origin 2000. The performance improvement arising from the use of the mechanisms is shown by the shaded portion of each bar.

The measurements in the figure show that together, the view-caching and load-balancing mechanisms improve performance across a range of applications, processor configurations, communication architecture interfaces, and target platforms. The mechanisms result in speedup improvements of up to 10 times over a baseline version of the application program. These improvements enable each application to achieve performance comparable with the best achievable via low-level programming means [169–172] (see Sections 7.2—7.5 for details), demonstrating the *collective sufficiency* of the mechanisms.

In addition, as measurements in Sections 7.2—7.5 on the Origin show, the mechanisms enable application performance on simpler communication architectures such as MSG, PUT/GET, and LD/ST to approach application performance on a cache-coherent shared memory machine, suggesting that the mechanisms can *compensate for the deficit of hardware communication features*. For the radiosity and Gröbner applications, performance on communication architectures without cache coherence actually exceeds that on the CC-LD/ST architecture. For IC-Cedar and phylogeny, performance on non-cache-coherent architectures is limited by the software overheads of access control, a situation that can be alleviated using finer grained access control methods [107, 108].

Individual impact. Figure 7.26 shows the individual performance impact of the view-caching mechanisms for each application on the Cray T3D and the SGI Origin 2000. The performance loss resulting from disabling the data locality mechanisms are shown by the shaded portion of the bar. Figure 7.27 shows similar graphs to highlight the individual performance impact of load balancing mechanisms.

The measurements in Figures 7.26 and 7.27 show that disabling either of the view-caching or load balancing-mechanisms can decrease achieved performance by up to 10 times. These results demonstrate that both the mechanisms are *individually effective*.

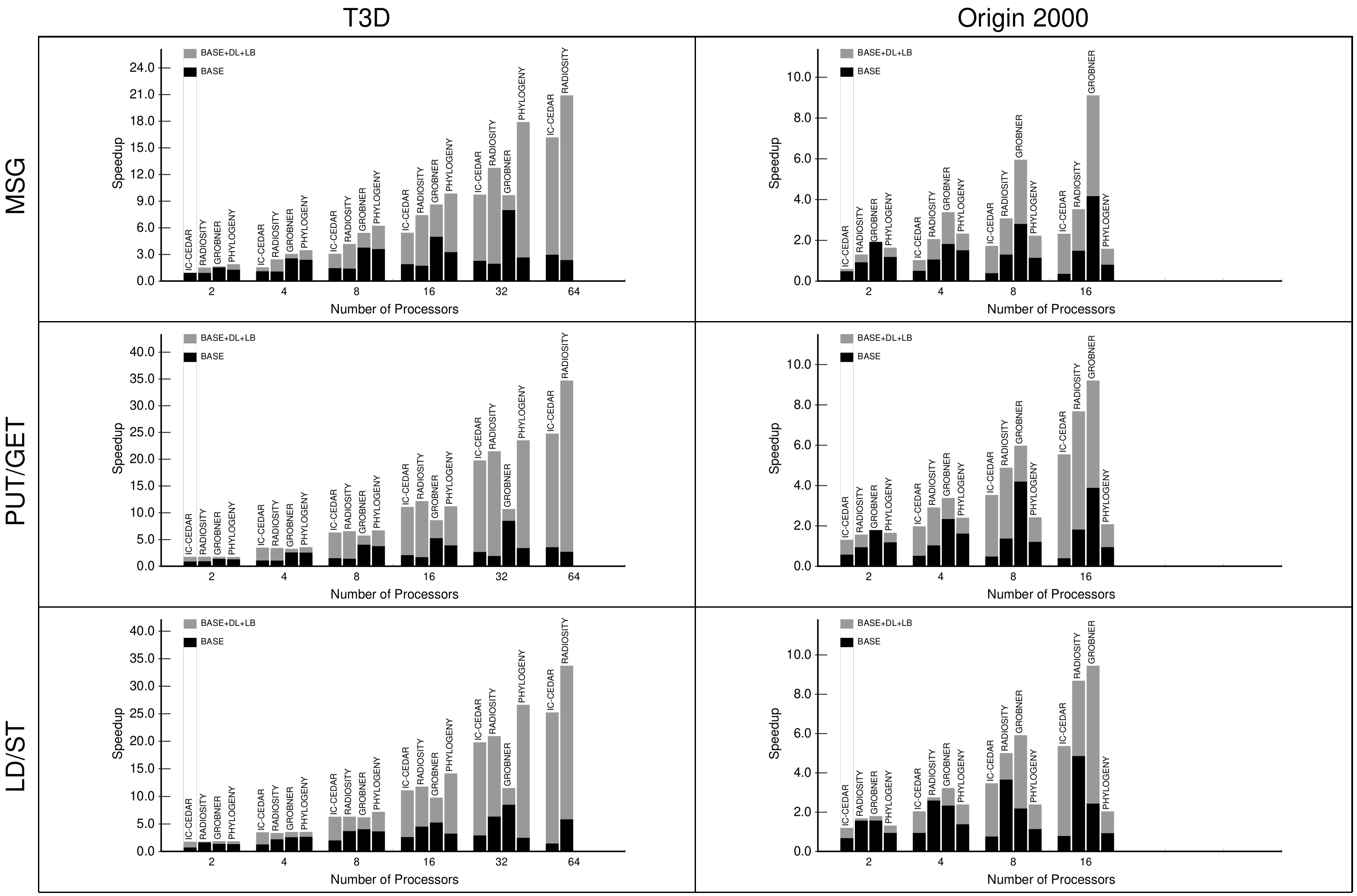

Figure 7.25 Aggregate performance impact of run-time mechanisms across all applications. The performance improvement arising from the mechanisms is shown by the shaded portion of the bar.

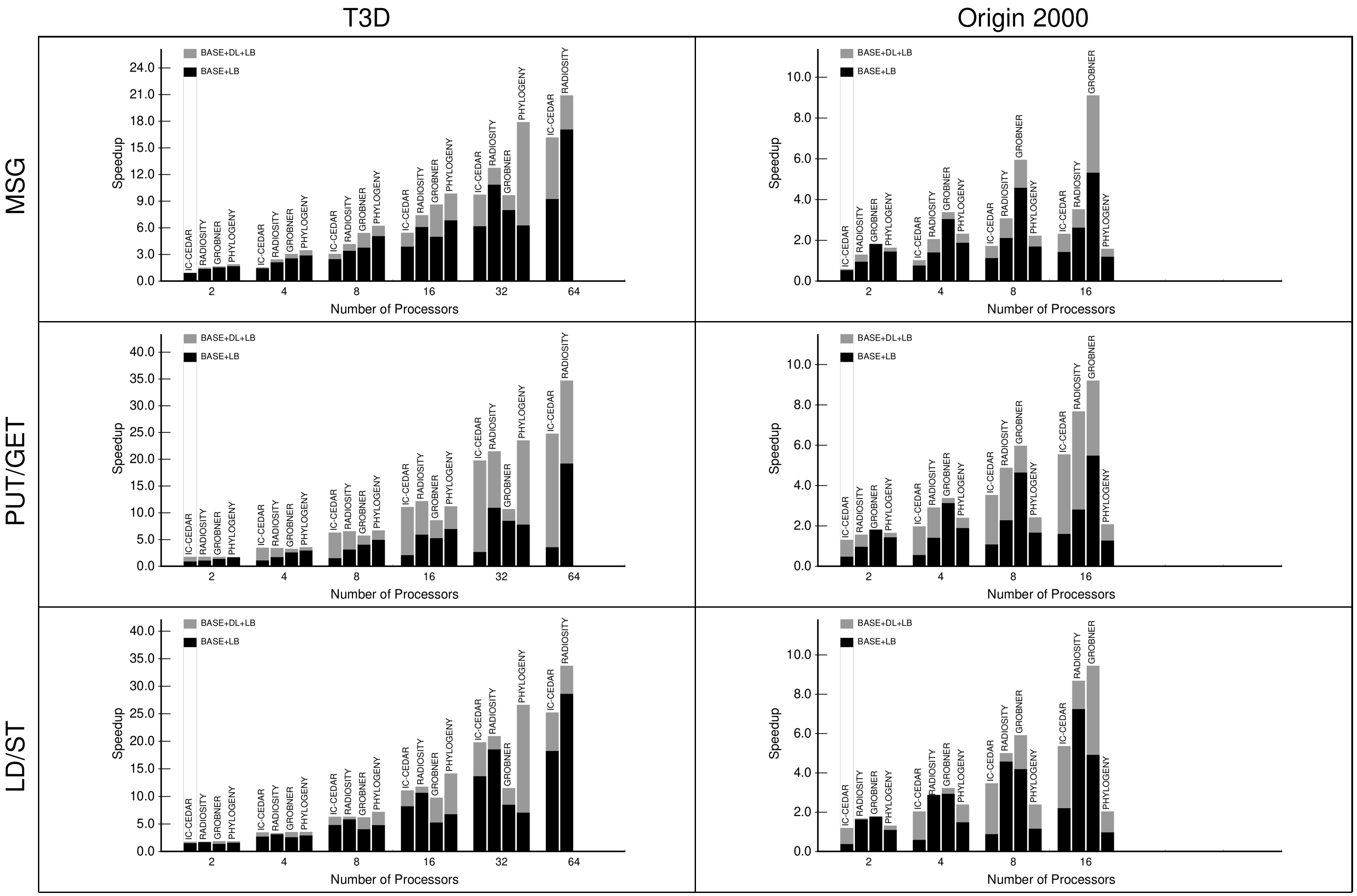

Figure 7.26 Individual performance impact of data locality mechanisms across all applications. The performance loss resulting from disabling the data locality mechanisms is shown by the shaded portion of the bar.

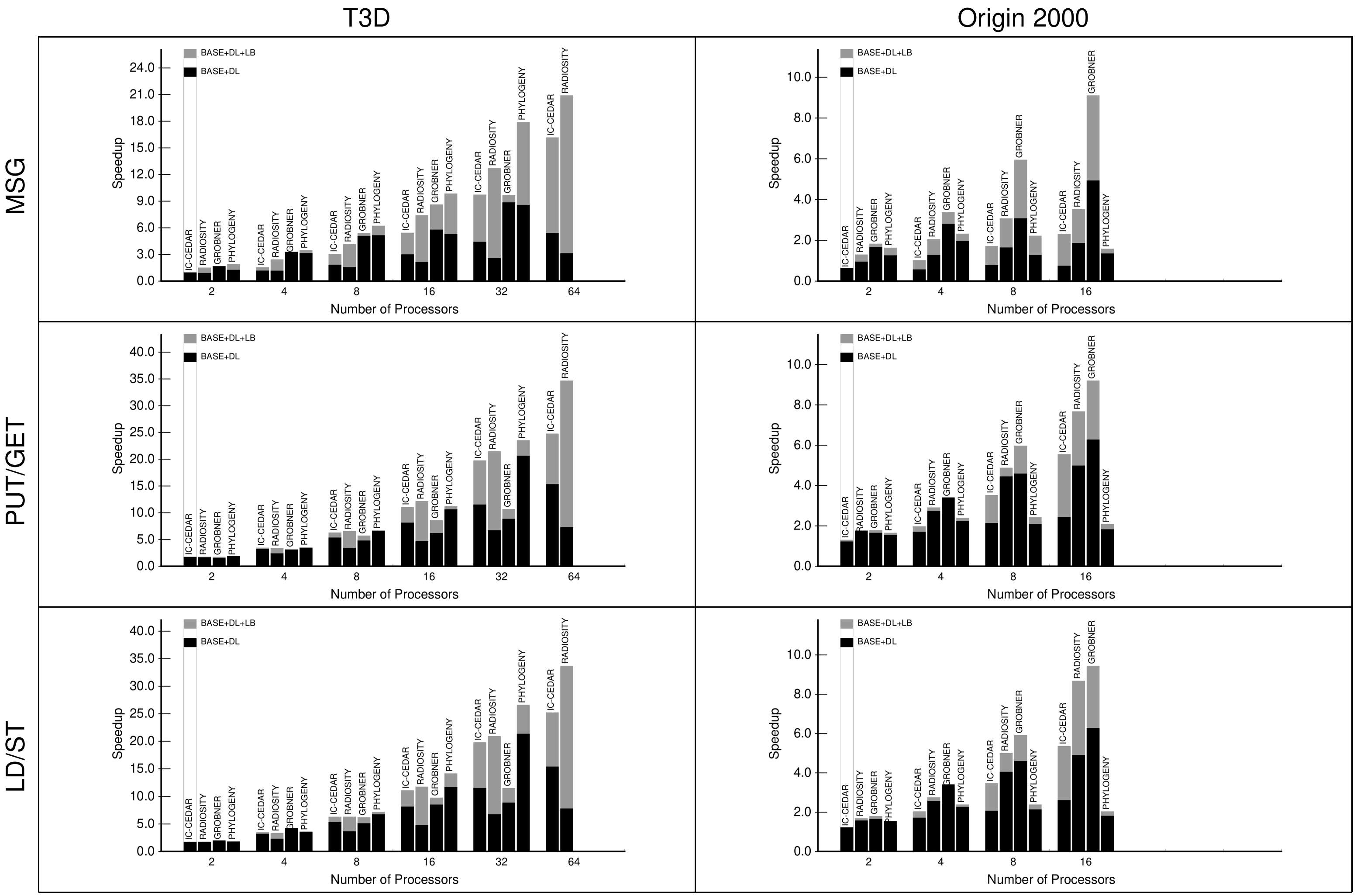

Figure 7.27 Individual performance impact of load-balancing mechanisms across all applications. The performance loss resulting from disabling the load-balancing mechanisms is shown by the shaded portion of the bar.

Detailed results in Sections 7.2—7.5 show that view caching mechanisms improve performance across the entire range of communication architectures. This improvement stems from three reasons:

1. Idle time reductions due to the use of asynchronous coherence protocols which require reduced remote processor involvement and synchronization.

2. Coherence overhead reductions due to detached views and protocols which eliminate false sharing.

3. Idle time reductions due to reduced concurrency control delays as in the Gröbner and phylogeny applications. These reductions come about because view caching protocols permit the overlap of detached read views with modifications to a shared object.

Our measurements also show that the benefits from view-caching protocols are most pronounced on PUT/GET architectures. Unlike synchronous object-consistent DSM protocols, decoupled and asynchronous view caching protocols are able to take advantage of one-sided hardware communication primitives. On the other hand, both sets of protocols utilize the MSG and LD/ST architectures in similar fashion.

The results in Sections 7.2—7.5 also show that load balancing mechanisms improve performance across the entire range of communication architectures. This improvement stems from three reasons:

1. Coherence time reductions due to locality-sensitive thread distribution as in the IC-Cedar and phylogeny applications.

2. Idle time reductions due to a balanced thread distribution as in the radiosity application.

3. Compute time reductions due to less redundant work created by priority-sensitive thread distribution as in the Gröbner and phylogeny applications.

7.7 Summary

In this chapter we have described an application-based performance evaluation of the view-caching and load-balancing frameworks described in Chapters 5 and 6 respectively.

Our evaluation was based upon four applications—IC-Cedar, radiosity, Gröbner, and phylogeny—which exhibit several dimensions of irregularity including unbalanced thread creation, wide variations in thread granularity, object access patterns which evolve with the computation, and shared objects that are concurrently read and modified. These application characteristics imply that careful management of data locality and load balancing is required for performance. However, the irregular nature prevents high performance even with low-overhead data locality techniques such as object-consistent DSM and load balancing techniques such as thread distribution via object placement.

The view-caching and load-balancing frameworks can alleviate these performance shortcomings. The view-caching framework exploits knowledge of unchanging object state, usability of inconsistent state, and information about concurrent activities against a shared object to construct custom coherence protocols for each of the applications which require reduced remote processor involvement and less synchronization. The load-balancing framework specifies custom load balancing policies for selected subsets of threads to achieve locality-sensitive and priority-sensitive balanced thread distribution.

Application performance results for various communication architectures on the Cray T3D and the SGI Origin 2000 demonstrate that the two mechanisms improve performance over the baseline version by up to ten times and are collectively sufficient, enabling a high-level expression of the program to achieve performance comparable to the best obtained via low-level programming means. In addition, the mechanisms enable performance on simpler communication architectures such as MSG, PUT/GET, and LD/ST to approach that on the CC-LD/ST architecture, suggesting that the run-time mechanisms can compensate for the deficit of hardware communication features.

The view-caching and load-balancing mechanisms are also individually effective: disabling either decreases achieved speedup by up to 10 times. View caching mechanisms improve performance by reducing coherence time and idle time, both due to the use of asynchronous protocols and due to reductions in concurrency control delays which come about because view caching protocols permit the overlap of detached read views with modifications to a shared object. The improvements from view caching protocols are most pronounced on the PUT/GET architecture because unlike synchronous object-consistent DSM protocols, asynchronous view-caching protocols are able to take advantage of one-sided hardware communication primitives. Load-balancing mechanisms yield performance benefits that arise from coherence time reductions due to locality-sensitive thread distribution, idle time reductions due to balanced thread distributions, and compute time reductions due to priority-sensitive thread distribution. These improvements are relatively insensitive to the underlying communication architecture.

CHAPTER 8

CONCLUSION

In this chapter, we summarize the research described in this thesis. Section 8.1 highlights the key research contributions. A perspective on these results is provided in Section 8.2, and we conclude in Section 8.3 by discussing avenues for future work.

8.1 Thesis Summary

Programming models based on dynamic multithreading enable convenient expression of irregular parallel applications which utilize sophisticated adaptive and hierarchical algorithmic cores and pointer-based data structures. These programming models provide high-level features such as object-oriented programming, a global object namespace, and elective concurrency which enable the natural specification of irregular computations. Despite these programmability advantages, such models have not gained widespread use because they are challenging to implement efficiently on scalable parallel machines. The challenges arise from properties of both the computational model and irregular computations which produce dynamic, irregular thread structure (e.g., granularity variations), and unpredictable data access patterns.

In this dissertation, we have described a run-time execution framework for dynamic multithreaded computations on scalable parallel machines which overcomes these challenges, delivering application performance comparable to the best achievable using low-level techniques. This framework, developed in the context of the Illinois Concert System, addresses the overall efficiency problem by separating concerns of local and parallel efficiency, enabling techniques to be independently developed and optimized for each. Our approach relies on local efficiency mechanisms to deliver performance conditioned upon run-time locality and load balance, leaving it up to the parallel efficiency mechanisms to create good locality and load balance.

Our approach achieves high local efficiency in spite of fine-granularity threads and frequent, unpredictable communication operations using two low-overhead thread management and communication mechanisms, hybrid stack-heap execution and pull messaging. These mechanisms recognize and exploit good locality and load balance situations at run time, increasing significantly the range of application behaviors which can be efficiently supported. In addition, they provide an effective building block for parallel efficiency, enabling development of mechanisms for the latter without concern for local thread and communication scheduling and aggregation. Hybrid stack heap execution is an efficient thread management mechanism which reduces overheads for fine-granularity threads despite lack of static information by exploiting run-time data locality to coalesce computation into coarse-grained threads on the fly. Pull messaging is a robust communication mechanism which exploits distributed message queuing and lazy receiver-initiated data transfer to achieve high performance decoupled from local processor activity and any imbalance in communication traffic pattern. Evaluation of these two mechanisms using a synthetic microkernel shows that when applicable, each mechanism individually improves performance by up to 50%, with stack heap execution improving performance for applications with fine granularity threads, and pull messaging improving performance for applications that communicate frequently. Together, they achieve robust high-performance for a range of application behaviors.

We achieve high parallel efficiency in spite of computation irregularity using robust data locality and load balancing mechanisms which exploit knowledge of application semantics and properties of the underlying target platform for performance. Our data locality framework, view caching, overcomes the performance disadvantages encountered even by state of the art object-consistent DSM implementations in the face of computation irregularity. A microkernel evaluation of object-consistent DSM shows that it exhibits high sensitivity to processor unresponsiveness and unbalanced sharing, achieving good performance only with supportive communication architectures that provide responsive, atomic operations against remote memory (e.g., LD/ST). Despite the availability of responsive operations in less supportive architectures such as PUT/GET, existing protocols are not able to exploit these characteristics for a performance advantage. On the other hand, view caching utilizes knowledge of application data access-semantics to construct latency-tolerant custom coherence protocols which reduce synchronization and increase tolerance to computation irregularity. Microkernel results show that these advantages contribute to improved performance (due to reduced coherence overheads and idle times) which is robust over processor unresponsiveness and unbalanced sharing, even on less supportive architectures such as MSG and PUT/GET. In addition, view caching protocols on a particular communication architecture achieve performance parity with object-consistent DSM on a more supportive architecture, indicating that robust run-time techniques can often compensate for a deficit of hardware support.

Our load balancing framework simultaneously enables thread distribution which is low overhead and yet sophisticated enough to preserve data locality and thread execution priorities. Motivated by intractability results for two formulations of the load balancing problem, our framework organizes object and thread placement heuristics into a cost hierarchy. Instead of dictating the use of one load balancing policy for all threads in the computation, our approach relies on a compiler or a programmer to select the most efficient policy for each set of threads. Application load balance is synthesized by composing these individual policies using a run-time custom scheduler framework. Microkernel evaluation of the suitability of different policies for different regions of the application space shows that no single policy provides best performance over the entire space, supporting an approach which allows each set of threads that behaves differently to be subject to a different load balancing policy. Our results show that object placement mechanisms are suitable when applications exhibit low imbalance or small thread granularities. For higher thread granularity and higher degrees of imbalance, thread placement policies are appropriate. Finally, when the application also requires prioritized thread execution, thread ordering mechanisms yield best overall performance.

Evaluation of the parallel efficiency mechanisms using four large irregular applications (IC-Cedar, radiosity, Gröbner, and phylogeny) demonstrates their collective sufficiency: Each application achieves performance on both the T3D and the Origin that is comparable to the best achievable via low-level programming means on scalable parallel systems. Specifically, our mechanisms together yield an improvement in application performance of up to 10 times over existing data locality and load balancing techniques. In addition, the mechanisms enable performance on less supportive communication architectures such as MSG, PUT/GET, and LD/ST to approach that on the CC-LD/ST architecture, suggesting that the run-time mechanisms can compensate for a lack of hardware support. Both view-caching and load-balancing frameworks are also individually effective: disabling either results in a speedup degradation of up to 10 times. View caching mechanisms improve performance by reducing coherence time and idle time, both due to the use of asynchronous protocols and due to reductions in concurrency control delays from overlap of detached read views with modifications to a shared object. Load balancing mechanisms yield performance benefits arising from coherence time reductions due to locality-sensitive thread distribution, idle time reductions due to balanced thread distributions, and compute time reductions due to priority-sensitive thread distribution.

8.2 Perspective

While our techniques for local and parallel efficiency were developed in the context of a specific concurrent object-oriented language, ICC++, the core set of mechanisms (thread management, communication, object caching, and load balancing) are language independent and provide the enabling implementation technology for a multitude of high-level languages on scalable parallel machines.

The main implication of our research is that there are no fundamental impediments to executing high-level programming languages efficiently on scalable parallel machines providing direct access to a low-latency, high-bandwidth communication substrate. Our results demonstrate that on such platforms, appropriate compiler and run-time techniques enable irregular applications expressed in a dynamic multithreaded computation style to achieve performance comparable to the best achievable using low-level means. Additionally, these run-time techniques result in significant performance improvements across a range of communication architectures, attesting to their general utility. These results facilitate increased utilization of parallel machines for general-purpose applications expressed in a high-level language by eliminating programmer management of computation granularity, distribution, and low-level concurrency control.

An additional implication is that run-time techniques for dynamic computations need to be designed very differently from traditional techniques for thread management, communication, data locality or load balance. Specifically, these techniques must be able to adapt to the dynamic nature of computation, and provide performance robust over computation irregularity. Unfortunately, most existing techniques do not satisfy these objectives because of the limited demands of the data parallel and message passing programming models. Our solution satisfies these objectives by adopting an integrated "systems" approach which involves several pieces: the compiler (for the hybrid stack-heap execution model), robust local efficiency techniques (e.g., pull messaging), novel parallel efficiency techniques which are able to leverage features of underlying communication architectures (e.g., view caching), and finally the knowledge of application semantics (in both the view caching and custom load balancing frameworks).

The integrated solution is a reflection of the fact that overcoming the disadvantages of the dynamic nature and unpredictability requires a closer cooperation between different components of the system than would be required for more regular computations.

To place our solution in context with related work, we review the main aspects of our approach. Our approach provides robust local and parallel efficiency mechanisms for dynamic multithreaded computations on scalable parallel machines (based on stock hardware) which exploit knowledge of application semantics to achieve performance for irregular applications that is comparable with the best reported in literature using low-level approaches. Thus, there are four comparisons of interest, which are discussed below.

Programming systems for irregular applications. Although there is a wealth of research on programming irregular parallel applications, none of these approaches provide a single programming interface while supporting all three dimensions of irregularity: data, control, and concurrency. Research efforts have largely focused on building special libraries which provide run-time support for one of several dimensions of irregularity. For example, the inspector-executor approach uses a library to support iterated irregular communication (such as in finite element simulations) [174]. The Multipol library provides a set of tools to build irregular applications, but no uniform high level interface [117]. Libraries such as Sparspak, Kelp, and A++/P++ support particular classes of algorithms on sparse matrices and adaptive meshes [180,181]. These systems have demonstrated good performance on large-scale irregular problems by exploiting specific properties of the restricted problem subset.

In contrast, our solution reflects a major philosophical difference in approach. Unlike the above systems which utilize a restricted, relatively low-level programming model and obtain efficiency by relying on hardwired decisions, we choose to adopt a general high-level programming model and tune its implementation for performance.

Hardware solutions for fine-grained computations. Several researchers have argued for including custom hardware capabilities such as multiple hardware contexts [133, 134], tightly integrated communication [134, 141, 182, 183], integrating thread creation with message reception [135, 136], and support for cache coherence [128,129,184,185] on the grounds that such support is essential for obtaining efficiency on fine-grained problems.

While it cannot be denied that some problems require low-overhead hardware support, a large class of applications can be efficiently executed without any specialized support using an approach similar to ours. Our mechanisms leverage a close coupling with the compiler and the use of application knowledge to compensate for the lack of a richer set of hardware primitives.

Implementations of fine-grained parallel languages. As we have stated, the run-time mechanisms reported in this dissertation can be viewed as part of the core enabling technology required to implement any high-level language on scalable parallel machines. Several other researchers have also developed mechanisms towards this end. Systems such as the Threaded Abstract Machine (TAM) model [8], pSather [33], Prelude [96], Olden [9], and Cilk [7] contain low-overhead communication, thread, and load balancing mechanisms, while systems such as Shasta [108] and Tempest [107] contain low-overhead data locality mechanisms.

The main difference of our mechanisms is the fact that they yield robust performance in the face of computation irregularity. As we saw in Chapters 4 and 5, mechanisms which are otherwise low-overhead can suffer significant performance degradation due to computation irregularity.

Exploiting application knowledge. A central theme in our parallel efficiency mechanisms is the use of knowledge of application semantics to offset performance degradation due to computation irregularity. While other researchers have eschewed the use of such information, relying instead of application oblivious data locality [77, 79, 107, 108] and load balancing techniques [121, 122], none of these systems have published results showing good performance with irregular applications. We believe that knowledge of application semantics is a requirement in the latter context and must be exploited by the language implementation to improve efficiency. Although we currently require programmer annotations to specify this knowledge, ideally one would prefer the appropriate application knowledge to automatically be extracted either using compiler analyses or adaptive profiling-based postmortem analyses.

8.3 Future Work

The research described in this dissertation focused primarily on demonstrating the feasibility of developing low-overhead, robust run-time techniques which would enable high-level dynamic multi-threaded computations to achieve performance comparable to the best reported in literature using low-level means. While we believe that we were successful in meeting this goal, many more advances are required to ensure the widespread use of parallel machines by a more general class of applications.

These advances span all levels from theoretical models of parallel computation all the way down to the architecture of target platforms. We briefly discuss these avenues for future work below.

Analysis of locality-aware load balancing strategies. The modeling of the locality-sensitive load balancing problem in Section 6.2 is only an initial step in bridging the gap between theory and practice. Both the load balancing model and analysis of load balancing strategies can be extended in the following ways:

1. *Designing approximation algorithms:* A natural extension to the NP-completeness result of the LB-CACHE formulation in Section 6.2 is to construct an approximation algorithm with provable performance bounds. Given the relationship of the LB-CACHE formulation to the traveling salesperson problem, a good starting point may be to look at good approximate solutions to that problem.

2. *Designing online algorithms:* Since our load balancing strategies deal with dynamic thread creation, it would be desirable to analyze the competitive ratio of these strategies to understand how far they are from the optimal. While the analysis of randomized work-sharing [121] and work-stealing [122] strategies are good starting points, they do not model data locality in the computation. The work on dag-consistency [161] is a promising step in the latter direction, but is still far from modeling general locality.

3. *Modeling realistic cache behavior:* The specific cache model did not affect the NP-completeness proof in Section 6.2, however it is likely to affect the design of online and approximation algorithms. It would be desirable to model both cache associativity and replacement behaviors such as LRU in the sequential case, and realistic coherence behavior in the parallel case.

4. *Adapting results to other problems:* Finally, our locality-aware load balancing formulation also has applications to other related problems where scheduling of tasks affects costs of data transfer. Examples include the problem of moving pages from disk to main memory in the context of virtual memory systems, and staging I/O or messaging operations using a fixed amount of buffering. It would be desirable to see if specific constraints of these different problems enable an optimal solution, or at least a better bound.

Automating view and policy choice. A natural extension of work done for this dissertation is to replace programmer selection of view-caching protocols and load-balancing policies with a more automatic selection procedure. Automating the process enables custom protocols and policies to be generated on a per-application basis. Two approaches are worth exploring: static compiler analyses of application structure, and profile-based postmortem analyses. In both cases, two steps are required: obtaining the necessary application information, and selecting a suitable policy.

Both approaches must first extract information about thread structure and object access behavior summarized for each thread creation site and object access site respectively. Examples of knowledge about thread structure include information about the amount of imbalance, granularity variations, and sensitivity to locality and priority. Examples of knowledge about object access include information about the subset of object state which is accessed, the access mode (e.g., read-only) in a particular phase of the computation, and interactions between specific types of threads. Section 7.6.2 identifies the specific kinds of information that must be obtained in order to effectively utilize the view caching and load balancing frameworks. Since the thread structure properties are data-dependent, a profile-based approach may have more success. In contrast, properties about object access are for the most part data-independent (with the exception of information about utility of inconsistent state), indicating that static compiler approaches may be suitable.

Once the above information is available, an appropriate policy must be chosen for each object access or thread creation site. Since multiple policies may be applicable at each site or there are constraints between sites (e.g., for view caching protocols, constraints are imposed by all sites which access the same object), a decision framework is required to choose the most suitable policy. Such a framework would utilize the selection guidelines described in Sections 5.5.3 and 6.6.5. Finally, the above choices may need to be revised based upon execution behavior. The revision could happen while the program is executing or for additional program runs.

Good starting points for the compiler-based approach are work on aliasing and structure analysis [63, 186–189], nonconcurrency analysis [190], commutativity analysis [191], and dynamic pointer alignment [69]. These describe compiler analyses for extracting information about object structure, thread interactions, their effect on object state, and the nature of object access respectively. Postmortem analysis has been explored in the context of optimizing object placement [192] and for providing performance feedback [193] in the Charm system.

Run-time mechanisms for diverse environments. Our research has been conducted in the context of dedicated parallel machines.[1] However, given that several diverse platforms are becoming popular, additional work is required to determine if the mechanisms described here are still suitable on these new platforms, or if new mechanisms are required. Two environments are of particular interest:

1. *Clusters of workstations:* These environments differ from dedicated parallel machines in several aspects: Communication costs are higher, and there is looser coupling between processors because of multitasking and the absence of gang-scheduling. These characteristics produce the same consequence as irregular computations, resulting in processors appearing unresponsive to incoming messages. Given this similarity, an interesting study would be to explore the advantages of view caching protocols in the context of a multitasking cluster. The research question that such a study must answer is whether custom view caching protocols can efficiently support a shared object space, overcoming the disadvantages of higher communication overheads and noncoordinated scheduling? Since alternate approaches have been proposed to limit uncoordinated process scheduling in cluster environments [194–197], it would be desirable to determine whether view caching is complementary to these approaches, or replaces them.

   The noncoordinated nature of process scheduling also has an impact on the load balancing policy chosen for a parallel job. An interesting avenue for future exploration is how the tradeoffs between different load balancing policies (e.g., work sharing and work stealing) change in the context of a cluster environment. Researchers [7] have argued that work stealing provides a more robust alternative for utilizing resources which can change dynamically, but additional work is required to validate this hypothesis.

2. *Shared memory multiprocessors:* Although we have evaluated the performance of our mechanisms on a hardware cache-coherent shared memory machine, the SGI Origin 2000, there is an additional issue which merits further investigation in similar environments. In this research, we restricted our attention to only two configurations on the Origin: where all processors in the machine were in the same address space, and where each processor was in its own address space. As our measurements on the Radiosity and Gröbner applications in Chapter 7 show, considering all processors as part of the same address space may not yield the best performance. Thus, an

[1] The Origin does support time sharing, but all experiments on larger processor configurations utilized a dedicated queue.

interesting study would be to explore the choice of when to use hardware versus software data locality mechanisms, and how this choice relates to application data access behavior.

Architectural primitives for shared memory. We have evaluated the effectiveness of four generic communication architecture interfaces (MSG, PUT/GET, LD/ST, and CC-LD/ST) in supporting efficient data locality and load-balancing mechanisms. These architectures were selected to systematically cover the space of available and announced scalable parallel machines. While our results show that view caching protocols make effective use of PUT/GET and LD/ST architectures, additional improvements might be possible with a communication architecture that provides a different set of underlying primitives.

Thus, an interesting future direction would be to look at designing an appropriate set of hardware primitives which would enable efficient, robust implementation of shared memory using view caching protocols. This is similar in spirit to the work on Pica [198] and Tempest [30, 185], both of which define a set of primitives for parallel processing. Based on the experiments reported in Chapter 7, an example set of primitives might include support for fast access control, put/get operations, and asynchronous and synchronous updates of global directory state. These primitives are quite different from those encountered in classic parallel machines, but have the potential of yielding significant performance advantages.

An aspect meriting additional study is building a hardware prototype that implements these primitives to characterize their performance advantages. Given the decreasing cost and increasing performance of reconfigurable logic (e.g., FPGAs), the logical next step would be to move part of the application-specific custom protocols into hardware. The open problem here is achieving an appropriate division of responsibility between the hardware and software components of the system.

APPENDIX A

RAW MICROKERNEL DATA

The microkernel data presented in Chapters 4, 5, and 6 were in the form of normalized execution times. For completeness, this appendix contains tables with the raw microkernel execution data.

A.1 Multiparty Communication Microkernel

Table A.1 lists the source overheads of push- and pull-messaging implementations on the Cray T3D for the multiparty communication microkernel described in Section 4.3. The data in the table are displayed graphically in Figure 4.9.

Table A.1 Source overheads (in μs) of push- and pull-messaging on the Cray T3D for various multiparty communication traffic patterns.

Push Messaging

MESSAGE	TRAFFIC PATTERNS					
SIZE (bytes)	pt-to-pt	all-to-all	random	hotspot,x4	hotspot,x16	all-to-one
16	4.91	6.33	6.45	6.44	8.27	34.47
64	5.13	7.25	8.86	8.82	15.51	52.54
256	6.65	11.57	18.51	18.61	29.33	140.13
1024	12.94	31.02	41.27	43.13	100.37	479.00

Pull Messaging

MESSAGE	TRAFFIC PATTERNS					
SIZE (bytes)	pt-to-pt	all-to-all	random	hotspot,x4	hotspot,x16	all-to-one
16	4.72	6.21	7.50	6.89	7.50	7.59
64	6.21	7.91	8.61	8.14	9.27	9.19
256	15.23	17.89	22.17	22.11	22.31	22.85
1024	43.04	49.31	77.44	77.71	77.34	80.72

A.2 Communicate-Compute Microkernel

Tables A.2 and A.3 list the execution efficiency on the SGI Origin 2000 and the Cray T3D, respectively, of the communicate-compute microkernel described in Chapter 4. The tables show measurements of various microkernel configurations (Base, Base+Pull, Base+Stack, and Base+Stack+Pull) for different values of compute granularity and percentage of local accesses. Measurements are on 16 processor machine configurations. The data in the tables are displayed graphically in Figure 4.2 and Figures 4.10–4.13.

Table A.2 Execution efficiency on the SGI Origin 2000 for various communicate-compute microkernel configurations.

Base

% Local Accesses	Compute Granularity (μs)					
	10.0	50.0	100.0	500.0	1000.0	5000.0
0	15.20	31.90	47.50	85.90	93.60	99.80
20	16.90	51.80	54.00	85.80	95.20	99.80
40	25.10	57.00	61.90	88.70	96.40	99.40
60	31.90	66.50	67.40	95.10	97.50	99.00
80	41.00	74.30	83.90	95.50	97.90	98.60
100	54.30	85.70	91.20	98.10	99.10	98.5

Base+Stack

% Local Accesses	Compute Granularity (μs)					
	10.0	50.0	100.0	500.0	1000.0	5000.0
0	21.50	43.00	67.50	90.70	95.10	99.0
20	40.40	56.90	72.90	92.40	96.10	99.1
40	45.40	68.90	77.30	94.90	97.00	99.4
60	53.70	71.40	83.00	96.70	98.00	99.6
80	58.50	81.80	89.40	97.70	98.80	99.8
100	85.70	92.10	97.70	99.90	99.90	100.0

Table A.3 Execution efficiency on the Cray T3D for various communicate-compute microkernel configurations.

Base

% Local Accesses	Compute Granularity (μs)					
	10.0	50.0	100.0	500.0	1000.0	5000.0
0	4.70	16.06	22.30	34.18	38.64	38.86
20	5.02	16.78	23.66	34.52	37.22	38.34
40	5.58	18.06	25.04	38.48	42.32	43.20
60	5.86	18.74	28.14	42.04	47.38	52.04
80	7.18	21.58	35.10	54.34	57.22	67.74
100	11.56	37.46	54.12	85.20	91.80	98.00

Base+Pull

% Local Accesses	Compute Granularity (μs)					
	10.0	50.0	100.0	500.0	1000.0	5000.0
0	6.80	25.00	39.70	76.50	86.60	96.90
20	7.30	26.50	41.60	77.80	87.40	97.00
40	7.80	28.00	43.50	79.00	88.20	97.20
60	8.60	30.20	46.00	80.70	89.20	97.40
80	9.30	32.00	47.80	81.50	88.70	95.30
100	10.50	34.90	51.50	83.80	91.00	97.80

Base+Stack

% Local Accesses	Compute Granularity (μs)					
	10.0	50.0	100.0	500.0	1000.0	5000.0
0	10.52	24.48	33.58	38.22	42.34	40.92
20	13.04	29.90	37.44	46.52	50.60	46.46
40	16.10	34.90	45.56	56.20	52.18	56.22
60	22.30	45.86	51.32	63.66	66.00	69.78
80	35.12	61.90	68.80	78.50	81.30	81.62
100	65.14	88.48	93.66	98.40	99.10	99.60

Base+Stack+Pull

% Local Accesses	Compute Granularity (μs)					
	10.0	50.0	100.0	500.0	1000.0	5000.0
0	14.60	42.10	58.70	87.30	93.20	98.50
20	18.20	48.10	64.80	89.90	94.60	98.80
40	22.10	54.40	70.00	91.80	95.60	99.00
60	29.20	63.50	77.20	94.10	96.90	99.20
80	41.30	74.80	85.20	96.40	98.00	99.50
100	69.50	90.60	94.80	98.70	99.30	99.70

A.3 Sharing Microkernel

Tables A.4 and A.5 list the execution times on the Cray T3D and the SGI Origin 2000, respectively, of the sharing microkernel described in Chapter 5.

The tables show measurements of three implementations of data locality support for different communication architectures on the two machines. The prefix of the configuration label denotes the data locality support: BASE-, ASYNC-, and SELF- correspond to traditional object-consistent DSM, and custom view coherence protocols READ_ASYNC and READ_SELF_INV respectively. The suffix of the configuration label denotes the communication architecture interface: MSG, PUT/GET, LD/ST, and CC-LD/ST correspond to the four communication architectures introduced in Section 3.4. For example, BASE-MSG lists the microkernel execution times when data locality support is implemented by object-consistent DSM protocols and the underlying architecture provides a MSG interface.

For each configuration, the tables list execution times for different values of microkernel parameters: d, o, τ, κ, and σ correspond to the access distribution, read-to-write ratio, use granularity, and the mean and standard deviation of the interrequest granularity, respectively. The TOTAL TIME, COMPUTE TIME, and IDLE TIME columns show the total execution time and its compute and idle time components respectively. The remaining portion of the execution time is attributable to coherence overheads. All measurements are on 32-node machine configurations. The data in the tables are displayed graphically in Figures 5.4–5.13.

Table A.4: Execution times on the Cray T3D (in seconds) for various sharing microkernel configurations.

BASE-MSG

d	o	τ	κ	σ	TOTAL TIME	COMPUTE TIME	IDLE TIME
random	95:5	10.0μs	10.0τ	0.0	0.89 +- 0.0004	0.61 +- 0.0002	0.19 +- 0.0101
hotspot,x4					0.95 +- 0.0006	0.61 +- 0.0000	0.23 +- 0.0069
hotspot,x8					0.98 +- 0.0000	0.61 +- 0.0000	0.27 +- 0.0118
random	95:5	10.0μs	10.0τ	0.0	0.89 +- 0.0004	0.61 +- 0.0002	0.19 +- 0.0101
	90:10				1.05 +- 0.0003	0.61 +- 0.0005	0.27 +- 0.0125
	85:15				1.21 +- 0.0000	0.61 +- 0.0000	0.39 +- 0.0130
random	95:5	1.0μs	10.0τ	0.0	0.34 +- 0.0001	0.11 +- 0.0001	0.14 +- 0.0104
		10.0μs			0.89 +- 0.0004	0.61 +- 0.0002	0.19 +- 0.0101
		100.0μs			6.62 +- 0.0059	5.66 +- 0.0020	0.85 +- 0.0127
random	95:5	10.0μs	0.1τ	0.0	0.35 +- 0.0001	0.11 +- 0.0000	0.15 +- 0.0099
			τ		0.37 +- 0.0002	0.16 +- 0.0001	0.13 +- 0.0098
			10.0τ		0.89 +- 0.0004	0.61 +- 0.0002	0.19 +- 0.0101

Table A.4: (continued)

random	95:5	10.0μs	10.0τ	0.0	0.89 +- 0.0004	0.61 +- 0.0002	0.19 +- 0.0101
				0.5κ	0.91 +- 0.0006	0.63 +- 0.0003	0.20 +- 0.0136
				κ	1.11 +- 0.0005	0.68 +- 0.0003	0.34 +- 0.0156

BASE-PUT/GET

d	o	τ	κ	σ	TOTAL TIME	COMPUTE TIME	IDLE TIME
random	95:5	10.0μs	10.0τ	0.0	0.79 +- 0.0003	0.60 +- 0.0003	0.13 +- 0.0062
hotspot,x4					0.83 +- 0.0004	0.60 +- 0.0003	0.16 +- 0.0055
hotspot,x8					0.85 +- 0.0003	0.60 +- 0.0003	0.18 +- 0.0068
random	95:5	10.0μs	10.0τ	0.0	0.79 +- 0.0003	0.60 +- 0.0003	0.13 +- 0.0062
	90:10				0.95 +- 0.0002	0.60 +- 0.0002	0.23 +- 0.0064
	85:15				0.95 +- 0.0003	0.60 +- 0.0003	0.21 +- 0.0082
random	95:5	1.0μs	10.0τ	0.0	0.22 +- 0.0001	0.09 +- 0.0000	0.06 +- 0.0062
		10.0μs			0.79 +- 0.0003	0.60 +- 0.0003	0.13 +- 0.0062
		100.0μs			6.52 +- 0.0034	5.65 +- 0.0020	0.80 +- 0.0064
random	95:5	10.0μs	0.1τ	0.0	0.22 +- 0.0001	0.09 +- 0.0000	0.06 +- 0.0062
			τ		0.33 +- 0.0002	0.14 +- 0.0000	0.13 +- 0.0071
			10.0τ		0.79 +- 0.0003	0.60 +- 0.0003	0.13 +- 0.0062
random	95:5	10.0μs	10.0τ	0.0	0.79 +- 0.0003	0.60 +- 0.0003	0.13 +- 0.0062
				0.5κ	0.88 +- 0.0004	0.61 +- 0.0005	0.21 +- 0.0121
				κ	0.90 +- 0.0000	0.67 +- 0.0003	0.19 +- 0.0142

BASE-LD/ST

d	o	τ	κ	σ	TOTAL TIME	COMPUTE TIME	IDLE TIME
random	95:5	10.0μs	10.0τ	0.0	0.74 +- 0.0002	0.59 +- 0.0000	0.02 +- 0.0072
hotspot,x4					0.74 +- 0.0004	0.59 +- 0.0005	0.02 +- 0.0054
hotspot,x8					0.78 +- 0.0003	0.59 +- 0.0005	0.04 +- 0.0077
random	95:5	10.0μs	10.0τ	0.0	0.74 +- 0.0002	0.59 +- 0.0000	0.02 +- 0.0072
	90:10				0.79 +- 0.0000	0.59 +- 0.0003	0.03 +- 0.0085
	85:15				0.83 +- 0.0006	0.59 +- 0.0004	0.03 +- 0.0108
random	95:5	1.0μs	10.0τ	0.0	0.23 +- 0.0001	0.08 +- 0.0000	0.02 +- 0.0076
		10.0μs			0.74 +- 0.0002	0.59 +- 0.0000	0.02 +- 0.0072
		100.0μs			5.79 +- 0.0034	5.64 +- 0.0024	0.01 +- 0.0067
random	95:5	10.0μs	0.1τ	0.0	0.24 +- 0.0001	0.08 +- 0.0000	0.02 +- 0.0076
			τ		0.28 +- 0.0001	0.13 +- 0.0001	0.02 +- 0.0071
			10.0τ		0.74 +- 0.0002	0.59 +- 0.0000	0.02 +- 0.0072
random	95:5	10.0μs	10.0τ	0.0	0.74 +- 0.0002	0.59 +- 0.0000	0.02 +- 0.0072
				0.5κ	0.75 +- 0.0002	0.60 +- 0.0002	0.03 +- 0.0120
				κ	0.80 +- 0.0004	0.66 +- 0.0002	0.03 +- 0.0148

ASYNC-MSG

d	o	τ	κ	σ	TOTAL TIME	COMPUTE TIME	IDLE TIME
random	95:5	10.0μs	10.0τ	0.0	0.88 +- 0.0003	0.61 +- 0.0004	0.16 +- 0.0089
hotspot,x4					0.93 +- 0.0002	0.61 +- 0.0003	0.19 +- 0.0069

Table A.4: (continued)

hotspot,x8					0.94 +- 0.0003	0.61 +- 0.0003	0.20 +- 0.0110
random	95:5	10.0μs	10.0τ	0.0	0.88 +- 0.0003	0.61 +- 0.0004	0.16 +- 0.0089
	90:10				1.00 +- 0.0006	0.61 +- 0.0003	0.22 +- 0.0109
	85:15				1.16 +- 0.0006	0.61 +- 0.0003	0.33 +- 0.0106
random	95:5	1.0μs	10.0τ	0.0	0.34 +- 0.0002	0.10 +- 0.0000	0.12 +- 0.0095
		10.0μs			0.88 +- 0.0003	0.61 +- 0.0004	0.16 +- 0.0089
		100.0μs			6.44 +- 0.0059	5.66 +- 0.0039	0.65 +- 0.0115
random	95:5	10.0μs	0.1τ	0.0	0.35 +- 0.0002	0.10 +- 0.0000	0.13 +- 0.0092
			τ		0.38 +- 0.0003	0.15 +- 0.0000	0.12 +- 0.0084
			10.0τ		0.88 +- 0.0003	0.61 +- 0.0004	0.16 +- 0.0089
random	95:5	10.0μs	10.0τ	0.0	0.88 +- 0.0003	0.61 +- 0.0004	0.16 +- 0.0089
				0.5κ	0.90 +- 0.0005	0.62 +- 0.0004	0.17 +- 0.0138
				κ	1.05 +- 0.0005	0.68 +- 0.0003	0.27 +- 0.0152

ASYNC-PUT/GET

d	o	τ	κ	σ	Total Time	Compute Time	Idle Time
random	95:5	10.0μs	10.0τ	0.0	0.73 +- 0.0005	0.60 +- 0.0002	0.03 +- 0.0042
hotspot,x4					0.74 +- 0.0003	0.60 +- 0.0002	0.03 +- 0.0036
hotspot,x8					0.74 +- 0.0003	0.60 +- 0.0005	0.03 +- 0.0046
random	95:5	10.0μs	10.0τ	0.0	0.73 +- 0.0005	0.60 +- 0.0002	0.03 +- 0.0042
	90:10				0.79 +- 0.0004	0.60 +- 0.0002	0.04 +- 0.0064
	85:15				0.82 +- 0.0004	0.60 +- 0.0003	0.04 +- 0.0057
random	95:5	1.0μs	10.0τ	0.0	0.22 +- 0.0000	0.09 +- 0.0000	0.03 +- 0.0046
		10.0μs			0.73 +- 0.0005	0.60 +- 0.0002	0.03 +- 0.0042
		100.0μs			5.80 +- 0.0052	5.65 +- 0.0024	0.05 +- 0.0065
random	95:5	10.0μs	0.1τ	0.0	0.22 +- 0.0001	0.09 +- 0.0001	0.03 +- 0.0046
			τ		0.27 +- 0.0000	0.14 +- 0.0001	0.03 +- 0.0045
			10.0τ		0.73 +- 0.0005	0.60 +- 0.0002	0.03 +- 0.0042
random	95:5	10.0μs	10.0τ	0.0	0.73 +- 0.0005	0.60 +- 0.0002	0.03 +- 0.0042
				0.5κ	0.75 +- 0.0002	0.61 +- 0.0003	0.04 +- 0.0116
				κ	0.81 +- 0.0005	0.67 +- 0.0004	0.06 +- 0.0142

ASYNC-LD/ST

d	o	τ	κ	σ	Total Time	Compute Time	Idle Time
random	95:5	10.0μs	10.0τ	0.0	0.74 +- 0.0004	0.59 +- 0.0002	0.02 +- 0.0070
hotspot,x4					0.74 +- 0.0004	0.59 +- 0.0005	0.02 +- 0.0054
hotspot,x8					0.78 +- 0.0003	0.59 +- 0.0005	0.04 +- 0.0077
random	95:5	10.0μs	10.0τ	0.0	0.74 +- 0.0004	0.59 +- 0.0002	0.02 +- 0.0070
	90:10				0.85 +- 0.0005	0.59 +- 0.0002	0.04 +- 0.0129
	85:15				0.89 +- 0.0003	0.59 +- 0.0003	0.05 +- 0.0145
random	95:5	1.0μs	10.0τ	0.0	0.26 +- 0.0001	0.08 +- 0.0000	0.03 +- 0.0105
		10.0μs			0.74 +- 0.0004	0.59 +- 0.0002	0.02 +- 0.0070
		100.0μs			5.81 +- 0.0020	5.64 +- 0.0037	0.02 +- 0.0082
random	95:5	10.0μs	0.1τ	0.0	0.27 +- 0.0002	0.08 +- 0.0000	0.03 +- 0.0105

Table A.4: (continued)

			τ		0.31 +- 0.0000	0.13 +- 0.0000	0.03 +- 0.0103
			10.0τ		0.74 +- 0.0004	0.59 +- 0.0002	0.02 +- 0.0070
random	95:5	10.0μs	10.0τ	0.0	0.74 +- 0.0004	0.59 +- 0.0002	0.02 +- 0.0070
				0.5κ	0.78 +- 0.0004	0.60 +- 0.0003	0.03 +- 0.0138
				κ	0.83 +- 0.0003	0.66 +- 0.0003	0.04 +- 0.0157

SELF-MSG

d	o	τ	κ	σ	TOTAL TIME	COMPUTE TIME	IDLE TIME
random	95:5	10.0μs	10.0τ	0.0	0.74 +- 0.0004	0.61 +- 0.0002	0.07 +- 0.0021
hotspot,x4					0.75 +- 0.0003	0.61 +- 0.0005	0.08 +- 0.0046
hotspot,x8					0.80 +- 0.0005	0.61 +- 0.0004	0.13 +- 0.0122
random	95:5	10.0μs	10.0τ	0.0	0.74 +- 0.0004	0.61 +- 0.0002	0.07 +- 0.0021
	90:10				0.75 +- 0.0002	0.61 +- 0.0002	0.07 +- 0.0019
	85:15				0.81 +- 0.0002	0.61 +- 0.0003	0.13 +- 0.0030
random	95:5	1.0μs	10.0τ	0.0	0.21 +- 0.0001	0.10 +- 0.0000	0.05 +- 0.0022
		10.0μs			0.74 +- 0.0004	0.61 +- 0.0002	0.07 +- 0.0021
		100.0μs			6.05 +- 0.0039	5.66 +- 0.0020	0.32 +- 0.0035
random	95:5	10.0μs	0.1τ	0.0	0.21 +- 0.0001	0.10 +- 0.0000	0.05 +- 0.0022
			τ		0.26 +- 0.0001	0.15 +- 0.0001	0.05 +- 0.0020
			10.0τ		0.74 +- 0.0004	0.61 +- 0.0002	0.07 +- 0.0021
random	95:5	10.0μs	10.0τ	0.0	0.74 +- 0.0004	0.61 +- 0.0002	0.07 +- 0.0021
				0.5κ	0.75 +- 0.0004	0.62 +- 0.0003	0.08 +- 0.0092
				κ	0.86 +- 0.0002	0.68 +- 0.0004	0.15 +- 0.0137

SELF-PUT/GET

d	o	τ	κ	σ	TOTAL TIME	COMPUTE TIME	IDLE TIME
random	95:5	10.0μs	10.0τ	0.0	0.67 +- 0.0005	0.60 +- 0.0003	0.02 +- 0.0012
hotspot,x4					0.67 +- 0.0004	0.60 +- 0.0002	0.02 +- 0.0012
hotspot,x8					0.67 +- 0.0003	0.60 +- 0.0003	0.02 +- 0.0012
random	95:5	10.0μs	10.0τ	0.0	0.67 +- 0.0005	0.60 +- 0.0003	0.02 +- 0.0012
	90:10				0.67 +- 0.0000	0.60 +- 0.0002	0.02 +- 0.0013
	85:15				0.67 +- 0.0004	0.60 +- 0.0004	0.03 +- 0.0014
random	95:5	1.0μs	10.0τ	0.0	0.17 +- 0.0000	0.09 +- 0.0001	0.03 +- 0.0021
		10.0μs			0.67 +- 0.0005	0.60 +- 0.0003	0.02 +- 0.0012
		100.0μs			5.72 +- 0.0034	5.65 +- 0.0020	0.02 +- 0.0011
random	95:5	10.0μs	0.1τ	0.0	0.17 +- 0.0001	0.09 +- 0.0000	0.03 +- 0.0022
			τ		0.22 +- 0.0001	0.14 +- 0.0000	0.03 +- 0.0021
			10.0τ		0.67 +- 0.0005	0.60 +- 0.0003	0.02 +- 0.0012
random	95:5	10.0μs	10.0τ	0.0	0.67 +- 0.0005	0.60 +- 0.0003	0.02 +- 0.0012
				0.5κ	0.68 +- 0.0000	0.61 +- 0.0004	0.03 +- 0.0090
				κ	0.73 +- 0.0004	0.67 +- 0.0004	0.05 +- 0.0137

Table A.4: (continued)

SELF-LD/ST

d	o	τ	κ	σ	Total Time	Compute Time	Idle Time
random	95:5	10.0μs	10.0τ	0.0	0.67 +- 0.0002	0.59 +- 0.0003	0.00 +- 0.0008
hotspot,x4					0.67 +- 0.0004	0.59 +- 0.0002	0.00 +- 0.0010
hotspot,x8					0.67 +- 0.0002	0.59 +- 0.0003	0.00 +- 0.0007
random	95:5	10.0μs	10.0τ	0.0	0.67 +- 0.0002	0.59 +- 0.0003	0.00 +- 0.0008
	90:10				0.67 +- 0.0003	0.59 +- 0.0002	0.00 +- 0.0007
	85:15				0.67 +- 0.0005	0.59 +- 0.0002	0.00 +- 0.0010
random	95:5	1.0μs	10.0τ	0.0	0.16 +- 0.0000	0.08 +- 0.0000	0.00 +- 0.0008
		10.0μs			0.67 +- 0.0002	0.59 +- 0.0003	0.00 +- 0.0008
		100.0μs			5.72 +- 0.0034	5.64 +- 0.0037	0.00 +- 0.0009
random	95:5	10.0μs	0.1τ	0.0	0.16 +- 0.0000	0.08 +- 0.0001	0.00 +- 0.0008
			τ		0.21 +- 0.0000	0.13 +- 0.0000	0.00 +- 0.0008
			10.0τ		0.67 +- 0.0002	0.59 +- 0.0003	0.00 +- 0.0008
random	95:5	10.0μs	10.0τ	0.0	0.67 +- 0.0002	0.59 +- 0.0003	0.00 +- 0.0008
				0.5κ	0.68 +- 0.0004	0.60 +- 0.0002	0.02 +- 0.0090
				κ	0.74 +- 0.0002	0.66 +- 0.0003	0.03 +- 0.0137

Table A.5: Execution times on the SGI Origin 2000 (in seconds) for various sharing microkernel configurations.

BASE-CC-LD/ST

d	o	τ	κ	σ	Total Time	Compute Time	Idle Time
random	95:5	10.0μs	10.0τ	0.0	0.59 +- 0.0005	0.56 +- 0.0005	0.00 +- 0.0034
hotspot,x4					0.58 +- 0.0008	0.56 +- 0.0005	0.00 +- 0.0028
hotspot,x8					0.57 +- 0.0008	0.56 +- 0.0004	0.01 +- 0.0028
random	95:5	10.0μs	10.0τ	0.0	0.59 +- 0.0005	0.56 +- 0.0005	0.00 +- 0.0034
	90:10				0.57 +- 0.0005	0.56 +- 0.0006	0.00 +- 0.0031
	85:15				0.59 +- 0.0013	0.57 +- 0.0009	0.01 +- 0.0039
random	95:5	1.0μs	10.0τ	0.0	0.08 +- 0.0002	0.07 +- 0.0002	0.01 +- 0.0033
		10.0μs			0.59 +- 0.0005	0.56 +- 0.0005	0.00 +- 0.0034
		100.0μs			5.54 +- 0.0068	5.53 +- 0.0060	0.01 +- 0.0085
random	95:5	10.0μs	0.1τ	0.0	0.07 +- 0.0004	0.07 +- 0.0002	0.00 +- 0.0032
			τ		0.12 +- 0.0003	0.11 +- 0.0002	0.00 +- 0.0030
			10.0τ		0.59 +- 0.0005	0.56 +- 0.0005	0.00 +- 0.0034
random	95:5	10.0μs	10.0τ	0.0	0.59 +- 0.0005	0.56 +- 0.0005	0.00 +- 0.0034
				0.5κ	0.60 +- 0.0008	0.58 +- 0.0008	0.01 +- 0.0103
				κ	0.64 +- 0.0007	0.63 +- 0.0007	0.01 +- 0.0106

Table A.5: (continued)

BASE-MSG

d	o	τ	κ	σ	TOTAL TIME	COMPUTE TIME	IDLE TIME
random	95:5	10.0μs	10.0τ	0.0	0.69 +- 0.0009	0.56 +- 0.0006	0.07 +- 0.0046
hotspot,x4					0.72 +- 0.0006	0.56 +- 0.0005	0.14 +- 0.0042
hotspot,x8					0.77 +- 0.0006	0.56 +- 0.0005	0.16 +- 0.0032
random	95:5	10.0μs	10.0τ	0.0	0.69 +- 0.0009	0.56 +- 0.0006	0.07 +- 0.0046
	90:10				0.74 +- 0.0009	0.56 +- 0.0005	0.10 +- 0.0062
	85:15				0.78 +- 0.0010	0.56 +- 0.0005	0.12 +- 0.0065
random	95:5	1.0μs	10.0τ	0.0	0.15 +- 0.0005	0.07 +- 0.0002	0.03 +- 0.0048
		10.0μs			0.69 +- 0.0009	0.56 +- 0.0006	0.07 +- 0.0046
		100.0μs			6.02 +- 0.0020	5.52 +- 0.0028	0.48 +- 0.0100
random	95:5	10.0μs	0.1τ	0.0	0.16 +- 0.0005	0.07 +- 0.0002	0.03 +- 0.0058
			τ		0.20 +- 0.0006	0.11 +- 0.0001	0.03 +- 0.0058
			10.0τ		0.69 +- 0.0009	0.56 +- 0.0006	0.07 +- 0.0046
random	95:5	10.0μs	10.0τ	0.0	0.69 +- 0.0009	0.56 +- 0.0006	0.07 +- 0.0046
				0.5κ	0.72 +- 0.0007	0.58 +- 0.0005	0.11 +- 0.0116
				κ	0.77 +- 0.0009	0.63 +- 0.0002	0.12 +- 0.0133

BASE-PUT/GET

d	o	τ	κ	σ	TOTAL TIME	COMPUTE TIME	IDLE TIME
random	95:5	10.0μs	10.0τ	0.0	0.67 +- 0.0009	0.56 +- 0.0006	0.07 +- 0.0041
hotspot,x4					0.70 +- 0.0014	0.56 +- 0.0006	0.09 +- 0.0049
hotspot,x8					0.71 +- 0.0010	0.56 +- 0.0006	0.12 +- 0.0054
random	95:5	10.0μs	10.0τ	0.0	0.67 +- 0.0009	0.56 +- 0.0006	0.07 +- 0.0041
	90:10				0.73 +- 0.0008	0.56 +- 0.0006	0.10 +- 0.0033
	85:15				0.76 +- 0.0013	0.57 +- 0.0007	0.12 +- 0.0071
random	95:5	1.0μs	10.0τ	0.0	0.14 +- 0.0007	0.07 +- 0.0002	0.02 +- 0.0029
		10.0μs			0.67 +- 0.0009	0.56 +- 0.0006	0.07 +- 0.0041
		100.0μs			6.01 +- 0.0048	5.53 +- 0.0039	0.47 +- 0.0064
random	95:5	10.0μs	0.1τ	0.0	0.14 +- 0.0006	0.07 +- 0.0002	0.03 +- 0.0061
			τ		0.19 +- 0.0008	0.11 +- 0.0002	0.03 +- 0.0060
			10.0τ		0.67 +- 0.0009	0.56 +- 0.0006	0.07 +- 0.0041
random	95:5	10.0μs	10.0τ	0.0	0.67 +- 0.0009	0.56 +- 0.0006	0.07 +- 0.0041
				0.5κ	0.71 +- 0.0007	0.58 +- 0.0006	0.10 +- 0.0120
				κ	0.79 +- 0.0011	0.63 +- 0.0005	0.14 +- 0.0155

BASE-LD/ST

d	o	τ	κ	σ	TOTAL TIME	COMPUTE TIME	IDLE TIME
random	95:5	10.0μs	10.0τ	0.0	0.64 +- 0.0008	0.56 +- 0.0007	0.04 +- 0.0114
hotspot,x4					0.61 +- 0.0007	0.56 +- 0.0008	0.02 +- 0.0057
hotspot,x8					0.64 +- 0.0009	0.56 +- 0.0007	0.03 +- 0.0050
random	95:5	10.0μs	10.0τ	0.0	0.64 +- 0.0008	0.56 +- 0.0007	0.04 +- 0.0114
	90:10				0.65 +- 0.0008	0.56 +- 0.0005	0.04 +- 0.0107

Table A.5: (continued)

	85:15				0.66 +- 0.0010	0.56 +- 0.0007	0.04 +- 0.0119
random	95:5	1.0μs	10.0τ	0.0	0.09 +- 0.0008	0.07 +- 0.0002	0.01 +- 0.0014
		10.0μs			0.64 +- 0.0008	0.56 +- 0.0007	0.04 +- 0.0114
		100.0μs			5.55 +- 0.0068	5.53 +- 0.0073	0.01 +- 0.0075
random	95:5	10.0μs	0.1τ	0.0	0.13 +- 0.0005	0.07 +- 0.0001	0.03 +- 0.0083
			τ		0.18 +- 0.0008	0.11 +- 0.0001	0.02 +- 0.0066
			10.0τ		0.64 +- 0.0008	0.56 +- 0.0007	0.04 +- 0.0114
random	95:5	10.0μs	10.0τ	0.0	0.64 +- 0.0008	0.56 +- 0.0007	0.04 +- 0.0114
				0.5κ	0.65 +- 0.0012	0.58 +- 0.0006	0.04 +- 0.0278
				κ	0.69 +- 0.0011	0.63 +- 0.0008	0.04 +- 0.0165

ASYNC-MSG

d	o	τ	κ	σ	TOTAL TIME	COMPUTE TIME	IDLE TIME
random	95:5	10.0μs	10.0τ	0.0	0.67 +- 0.0009	0.56 +- 0.0005	0.06 +- 0.0047
hotspot,x4					0.71 +- 0.0005	0.56 +- 0.0004	0.11 +- 0.0058
hotspot,x8					0.73 +- 0.0005	0.56 +- 0.0003	0.12 +- 0.0045
random	95:5	10.0μs	10.0τ	0.0	0.67 +- 0.0009	0.56 +- 0.0005	0.06 +- 0.0047
	90:10				0.72 +- 0.0010	0.56 +- 0.0005	0.08 +- 0.0078
	85:15				0.75 +- 0.0011	0.57 +- 0.0008	0.09 +- 0.0075
random	95:5	1.0μs	10.0τ	0.0	0.15 +- 0.0008	0.07 +- 0.0002	0.03 +- 0.0049
		10.0μs			0.67 +- 0.0009	0.56 +- 0.0005	0.06 +- 0.0047
		100.0μs			5.89 +- 0.0048	5.53 +- 0.0044	0.34 +- 0.0058
random	95:5	10.0μs	0.1τ	0.0	0.16 +- 0.0008	0.07 +- 0.0001	0.03 +- 0.0066
			τ		0.19 +- 0.0010	0.11 +- 0.0001	0.03 +- 0.0063
			10.0τ		0.67 +- 0.0009	0.56 +- 0.0005	0.06 +- 0.0047
random	95:5	10.0μs	10.0τ	0.0	0.67 +- 0.0009	0.56 +- 0.0005	0.06 +- 0.0047
				0.5κ	0.69 +- 0.0010	0.58 +- 0.0006	0.07 +- 0.0127
				κ	0.75 +- 0.0012	0.63 +- 0.0005	0.09 +- 0.0129

ASYNC-PUT/GET

d	o	τ	κ	σ	TOTAL TIME	COMPUTE TIME	IDLE TIME
random	95:5	10.0μs	10.0τ	0.0	0.60 +- 0.0007	0.56 +- 0.0005	0.01 +- 0.0031
hotspot,x4					0.60 +- 0.0009	0.56 +- 0.0005	0.01 +- 0.0034
hotspot,x8					0.60 +- 0.0005	0.56 +- 0.0005	0.01 +- 0.0025
random	95:5	10.0μs	10.0τ	0.0	0.60 +- 0.0007	0.56 +- 0.0005	0.01 +- 0.0031
	90:10				0.62 +- 0.0011	0.56 +- 0.0004	0.01 +- 0.0044
	85:15				0.63 +- 0.0011	0.56 +- 0.0005	0.01 +- 0.0041
random	95:5	1.0μs	10.0τ	0.0	0.10 +- 0.0004	0.07 +- 0.0001	0.01 +- 0.0019
		10.0μs			0.60 +- 0.0007	0.56 +- 0.0005	0.01 +- 0.0031
		100.0μs			5.56 +- 0.0024	5.53 +- 0.0083	0.01 +- 0.0037
random	95:5	10.0μs	0.1τ	0.0	0.11 +- 0.0006	0.07 +- 0.0001	0.01 +- 0.0024
			τ		0.15 +- 0.0007	0.11 +- 0.0001	0.01 +- 0.0026
			10.0τ		0.60 +- 0.0007	0.56 +- 0.0005	0.01 +- 0.0031
random	95:5	10.0μs	10.0τ	0.0	0.60 +- 0.0007	0.56 +- 0.0005	0.01 +- 0.0031

Table A.5: (continued)

				0.5κ	0.62 +- 0.0009	0.58 +- 0.0006	0.02 +- 0.0108
				κ	0.66 +- 0.0011	0.64 +- 0.0063	0.03 +- 0.0125

ASYNC-LD/ST

d	o	τ	κ	σ	TOTAL TIME	COMPUTE TIME	IDLE TIME
random	95:5	10.0μs	10.0τ	0.0	0.60 +- 0.0008	0.56 +- 0.0005	0.01 +- 0.0044
hotspot,x4					0.60 +- 0.0011	0.56 +- 0.0005	0.01 +- 0.0054
hotspot,x8					0.62 +- 0.0010	0.56 +- 0.0005	0.01 +- 0.0045
random	95:5	10.0μs	10.0τ	0.0	0.60 +- 0.0008	0.56 +- 0.0005	0.01 +- 0.0044
	90:10				0.62 +- 0.0017	0.56 +- 0.0002	0.01 +- 0.0058
	85:15				0.63 +- 0.0017	0.56 +- 0.0003	0.01 +- 0.0060
random	95:5	1.0μs	10.0τ	0.0	0.11 +- 0.0009	0.07 +- 0.0002	0.01 +- 0.0042
		10.0μs			0.60 +- 0.0008	0.56 +- 0.0005	0.01 +- 0.0044
		100.0μs			5.56 +- 0.0060	5.53 +- 0.0059	0.01 +- 0.0068
random	95:5	10.0μs	0.1τ	0.0	0.11 +- 0.0010	0.07 +- 0.0001	0.01 +- 0.0044
			τ		0.15 +- 0.0010	0.11 +- 0.0003	0.01 +- 0.0040
			10.0τ		0.60 +- 0.0008	0.56 +- 0.0005	0.01 +- 0.0044
random	95:5	10.0μs	10.0τ	0.0	0.60 +- 0.0008	0.56 +- 0.0005	0.01 +- 0.0044
				0.5κ	0.63 +- 0.0058	0.58 +- 0.0006	0.04 +- 0.0121
				κ	0.67 +- 0.0011	0.63 +- 0.0007	0.03 +- 0.0129

SELF-MSG

d	o	τ	κ	σ	TOTAL TIME	COMPUTE TIME	IDLE TIME
random	95:5	10.0μs	10.0τ	0.0	0.62 +- 0.0006	0.56 +- 0.0005	0.03 +- 0.0020
hotspot,x4					0.63 +- 0.0007	0.56 +- 0.0006	0.04 +- 0.0036
hotspot,x8					0.64 +- 0.0022	0.56 +- 0.0004	0.04 +- 0.0090
random	95:5	10.0μs	10.0τ	0.0	0.62 +- 0.0006	0.56 +- 0.0005	0.03 +- 0.0020
	90:10				0.61 +- 0.0005	0.56 +- 0.0005	0.03 +- 0.0016
	85:15				0.61 +- 0.0005	0.56 +- 0.0004	0.03 +- 0.0015
random	95:5	1.0μs	10.0τ	0.0	0.12 +- 0.0002	0.07 +- 0.0001	0.01 +- 0.0012
		10.0μs			0.62 +- 0.0006	0.56 +- 0.0005	0.03 +- 0.0020
		100.0μs			5.69 +- 0.0048	5.53 +- 0.0046	0.15 +- 0.0056
random	95:5	10.0μs	0.1τ	0.0	0.12 +- 0.0003	0.07 +- 0.0002	0.01 +- 0.0011
			τ		0.16 +- 0.0003	0.11 +- 0.0002	0.01 +- 0.0008
			10.0τ		0.62 +- 0.0006	0.56 +- 0.0005	0.03 +- 0.0020
random	95:5	10.0μs	10.0τ	0.0	0.62 +- 0.0006	0.56 +- 0.0005	0.03 +- 0.0020
				0.5κ	0.63 +- 0.0006	0.58 +- 0.0005	0.04 +- 0.0104
				κ	0.70 +- 0.0008	0.63 +- 0.0005	0.06 +- 0.0120

SELF-PUT/GET

d	o	τ	κ	σ	TOTAL TIME	COMPUTE TIME	IDLE TIME
random	95:5	10.0μs	10.0τ	0.0	0.59 +- 0.0007	0.56 +- 0.0005	0.00 +- 0.0004
hotspot,x4					0.60 +- 0.0008	0.56 +- 0.0008	0.00 +- 0.0009

Table A.5: (continued)

hotspot,x8					0.62 +- 0.0010	0.56 +- 0.0008	0.01 +- 0.0010
random	95:5	10.0μs	10.0τ	0.0	0.59 +- 0.0007	0.56 +- 0.0005	0.00 +- 0.0004
	90:10				0.59 +- 0.0008	0.56 +- 0.0005	0.01 +- 0.0009
	85:15				0.59 +- 0.0005	0.56 +- 0.0007	0.01 +- 0.0009
random	95:5	1.0μs	10.0τ	0.0	0.09 +- 0.0003	0.07 +- 0.0001	0.00 +- 0.0006
		10.0μs			0.59 +- 0.0007	0.56 +- 0.0005	0.00 +- 0.0004
		100.0μs			5.56 +- 0.0082	5.52 +- 0.0031	0.03 +- 0.0090
random	95:5	10.0μs	0.1τ	0.0	0.10 +- 0.0003	0.07 +- 0.0001	0.00 +- 0.0007
			τ		0.13 +- 0.0004	0.11 +- 0.0004	0.00 +- 0.0004
			10.0τ		0.59 +- 0.0007	0.56 +- 0.0005	0.00 +- 0.0004
random	95:5	10.0μs	10.0τ	0.0	0.59 +- 0.0007	0.56 +- 0.0005	0.00 +- 0.0004
				0.5κ	0.60 +- 0.0005	0.58 +- 0.0005	0.02 +- 0.0096
				κ	0.65 +- 0.0006	0.63 +- 0.0005	0.03 +- 0.0133

SELF-LD/ST

d	o	τ	κ	σ	TOTAL TIME	COMPUTE TIME	IDLE TIME
random	95:5	10.0μs	10.0τ	0.0	0.60 +- 0.0005	0.56 +- 0.0005	0.00 +- 0.0034
hotspot,x4					0.60 +- 0.0008	0.56 +- 0.0005	0.00 +- 0.0028
hotspot,x8					0.59 +- 0.0008	0.56 +- 0.0004	0.01 +- 0.0028
random	95:5	10.0μs	10.0τ	0.0	0.60 +- 0.0005	0.56 +- 0.0005	0.00 +- 0.0034
	90:10				0.59 +- 0.0005	0.56 +- 0.0006	0.00 +- 0.0031
	85:15				0.61 +- 0.0013	0.57 +- 0.0009	0.01 +- 0.0039
random	95:5	1.0μs	10.0τ	0.0	0.10 +- 0.0002	0.07 +- 0.0002	0.01 +- 0.0033
		10.0μs			0.60 +- 0.0005	0.56 +- 0.0005	0.00 +- 0.0034
		100.0μs			5.56 +- 0.0068	5.53 +- 0.0060	0.02 +- 0.0085
random	95:5	10.0μs	0.1τ	0.0	0.10 +- 0.0004	0.07 +- 0.0002	0.00 +- 0.0032
			τ		0.14 +- 0.0003	0.11 +- 0.0002	0.00 +- 0.0030
			10.0τ		0.60 +- 0.0005	0.56 +- 0.0005	0.00 +- 0.0034
random	95:5	10.0μs	10.0τ	0.0	0.60 +- 0.0005	0.56 +- 0.0005	0.00 +- 0.0034
				0.5κ	0.61 +- 0.0008	0.58 +- 0.0008	0.02 +- 0.0103
				κ	0.65 +- 0.0007	0.63 +- 0.0007	0.02 +- 0.0106

A.4 Task Generation Microkernel

Tables A.6 and A.7 list the execution times on the Cray T3D and the SGI Origin 2000, respectively, of the task generation microkernel described in Chapter 6.

The tables show measurements for five load-balancing policies on the two machines: DATA, WSHR, WSHR-LOC, WSTL, and WSTL-GBL configurations denote the `ObjectPlacement`, `WorkSharing`, `WorkSharingLocal`, `WorkStealing`, and `WorkStealingGlobal` policies, respectively. Additionally, the WSTL-LD/ST and WSTL-GBL-LD/ST configurations list the performance of the `WorkStealing` and `WorkStealingGlobal` policies, respectively, on the LD/ST communication architecture. As described in Chapter 6, only these two policies exhibit different implementations across the communication architectures.

For each configuration, the tables list execution times for different values of microkernel parameters: v, τ, l, and p correspond to the thread imbalance (κ = 1000), thread granularity, locality penalty, and priority penalty respectively. The TOTAL, COMPUTE, STRETCH, and IDLE columns show the total execution time and its compute time, stretch time, and idle time components respectively. Stretch time refers to the amount of redundant computation that must be performed when threads are executed out of priority order. The remaining portion of the execution time is attributable to runtime overheads. All measurements are on 32-node machine configurations. The data in the tables are displayed graphically in Figures 6.7–6.12.

Table A.6: Execution times on the Cray T3D (in seconds) for various task microkernel configurations.

DATA

v	τ	l	p	TOTAL	COMPUTE	STRETCH	IDLE
0.0	50.0μs	0.5τ	0.0	0.07 +- 0.0000	0.06 +- 0.0000	0.00 +- 0.0000	0.00 +- 0.0000
	100.0μs			0.13 +- 0.0000	0.11 +- 0.0001	0.00 +- 0.0000	0.00 +- 0.0000
	500.0μs			0.58 +- 0.0003	0.56 +- 0.0004	0.00 +- 0.0000	0.00 +- 0.0000
0.5κ	50.0μs	0.5τ	0.0	0.12 +- 0.0000	0.05 +- 0.0235	0.00 +- 0.0000	0.05 +- 0.0000
	100.0μs			0.27 +- 0.0000	0.13 +- 0.0540	0.00 +- 0.0000	0.12 +- 0.0000
	500.0μs			2.83 +- 0.0010	0.68 +- 0.3026	0.00 +- 0.0000	1.80 +- 0.0000
κ	50.0μs	0.5τ	0.0	0.22 +- 0.0001	0.08 +- 0.0464	0.00 +- 0.0000	0.12 +- 0.0000
	100.0μs			0.27 +- 0.0002	0.13 +- 0.1060	0.00 +- 0.0000	0.12 +- 0.0000
	500.0μs			1.88 +- 0.0005	0.81 +- 0.4639	0.00 +- 0.0000	1.04 +- 0.0000
0.5κ	500.0μs	0.0	0.0	1.24 +- 0.0008	0.63 +- 0.2233	0.00 +- 0.0000	0.59 +- 0.0000
		0.5τ		2.83 +- 0.0010	0.68 +- 0.3026	0.00 +- 0.0000	1.80 +- 0.0000
		τ		1.24 +- 0.0007	0.65 +- 0.2700	0.00 +- 0.0000	0.57 +- 0.0000
0.5κ	500.0μs	0.5τ	0.0	2.83 +- 0.0010	0.68 +- 0.3026	0.00 +- 0.0000	1.80 +- 0.0000

Table A.6: (continued)

			τ	2.11 +- 0.0007	0.52 +- 0.2690	0.19 +- 0.2008	1.17 +- 0.0000
			5τ	5.22 +- 0.0014	0.61 +- 0.2967	1.37 +- 1.1445	2.90 +- 0.0000

WSHR

v	τ	l	p	TOTAL	COMPUTE	STRETCH	IDLE
0.0	50.0μs	0.5τ	0.0	0.13 +- 0.0001	0.06 +- 0.0001	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.20 +- 0.0001	0.11 +- 0.0003	0.00 +- 0.0000	0.01 +- 0.0000
	500.0μs			0.86 +- 0.0006	0.56 +- 0.0010	0.00 +- 0.0000	0.01 +- 0.0000
0.5κ	50.0μs	0.5τ	0.0	0.12 +- 0.0001	0.05 +- 0.0001	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.24 +- 0.0001	0.13 +- 0.0003	0.00 +- 0.0000	0.01 +- 0.0000
	500.0μs			1.08 +- 0.0003	0.68 +- 0.0008	0.00 +- 0.0000	0.02 +- 0.0000
κ	50.0μs	0.5τ	0.0	0.17 +- 0.0001	0.08 +- 0.0001	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.24 +- 0.0001	0.13 +- 0.0002	0.00 +- 0.0000	0.01 +- 0.0000
	500.0μs			1.22 +- 0.0006	0.81 +- 0.0010	0.00 +- 0.0000	0.02 +- 0.0000
0.5κ	500.0μs	0.0	0.0	0.68 +- 0.0002	0.63 +- 0.0008	0.00 +- 0.0000	0.01 +- 0.0000
		0.5τ		1.08 +- 0.0003	0.68 +- 0.0008	0.00 +- 0.0000	0.02 +- 0.0000
		τ		1.28 +- 0.0005	0.65 +- 0.0012	0.00 +- 0.0000	0.03 +- 0.0000
0.5κ	500.0μs	0.5τ	0.0	1.08 +- 0.0003	0.68 +- 0.0008	0.00 +- 0.0000	0.02 +- 0.0000
			τ	1.12 +- 0.0005	0.52 +- 0.0009	0.24 +- 0.0266	0.07 +- 0.0000
			5τ	2.92 +- 0.0000	0.61 +- 0.0014	1.49 +- 0.1768	0.45 +- 0.0000

WSHR-LOC

v	τ	l	p	TOTAL	COMPUTE	STRETCH	IDLE
0.0	50.0μs	0.5τ	0.0	0.10 +- 0.0001	0.06 +- 0.0009	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.16 +- 0.0001	0.11 +- 0.0018	0.00 +- 0.0000	0.01 +- 0.0000
	500.0μs			0.67 +- 0.0003	0.56 +- 0.0088	0.00 +- 0.0000	0.02 +- 0.0000
0.5κ	50.0μs	0.5τ	0.0	0.10 +- 0.0000	0.05 +- 0.0013	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.19 +- 0.0001	0.13 +- 0.0014	0.00 +- 0.0000	0.02 +- 0.0000
	500.0μs			0.84 +- 0.0002	0.68 +- 0.0132	0.00 +- 0.0000	0.03 +- 0.0000
κ	50.0μs	0.5τ	0.0	0.13 +- 0.0000	0.08 +- 0.0014	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.19 +- 0.0001	0.13 +- 0.0026	0.00 +- 0.0000	0.02 +- 0.0000
	500.0μs			0.93 +- 0.0006	0.81 +- 0.0113	0.00 +- 0.0000	0.03 +- 0.0000
0.5κ	500.0μs	0.0	0.0	0.70 +- 0.0005	0.63 +- 0.0107	0.00 +- 0.0000	0.02 +- 0.0000
		0.5τ		0.84 +- 0.0002	0.68 +- 0.0132	0.00 +- 0.0000	0.03 +- 0.0000
		τ		0.82 +- 0.0003	0.65 +- 0.0124	0.00 +- 0.0000	0.03 +- 0.0000
0.5κ	500.0μs	0.5τ	0.0	0.84 +- 0.0002	0.68 +- 0.0132	0.00 +- 0.0000	0.03 +- 0.0000
			τ	0.97 +- 0.0002	0.52 +- 0.0097	0.26 +- 0.0325	0.09 +- 0.0000
			5τ	2.76 +- 0.0000	0.61 +- 0.0093	1.52 +- 0.1873	0.47 +- 0.0000

WSTL

v	τ	l	p	TOTAL	COMPUTE	STRETCH	IDLE
0.0	50.0μs	0.5τ	0.0	0.10 +- 0.0000	0.06 +- 0.0087	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.17 +- 0.0001	0.11 +- 0.0240	0.00 +- 0.0000	0.02 +- 0.0000

Table A.6: (continued)

	500.0μs			0.71 +- 0.0003	0.56 +- 0.1002	0.00 +- 0.0000	0.07 +- 0.0000
0.5κ	50.0μs	0.5τ	0.0	0.11 +- 0.0001	0.05 +- 0.0099	0.00 +- 0.0000	0.02 +- 0.0000
	100.0μs			0.25 +- 0.0001	0.13 +- 0.0226	0.00 +- 0.0000	0.07 +- 0.0000
	500.0μs			1.26 +- 0.0003	0.68 +- 0.1412	0.00 +- 0.0000	0.17 +- 0.0000
κ	50.0μs	0.5τ	0.0	0.18 +- 0.0001	0.08 +- 0.0165	0.00 +- 0.0000	0.04 +- 0.0000
	100.0μs			0.25 +- 0.0001	0.13 +- 0.0316	0.00 +- 0.0000	0.07 +- 0.0000
	500.0μs			1.13 +- 0.0005	0.81 +- 0.1576	0.00 +- 0.0000	0.17 +- 0.0000
0.5κ	500.0μs	0.0	0.0	0.78 +- 0.0005	0.63 +- 0.0857	0.00 +- 0.0000	0.10 +- 0.0000
		0.5τ		1.26 +- 0.0003	0.68 +- 0.1412	0.00 +- 0.0000	0.17 +- 0.0000
		τ		0.98 +- 0.0002	0.65 +- 0.1421	0.00 +- 0.0000	0.13 +- 0.0000
0.5κ	500.0μs	0.5τ	0.0	1.26 +- 0.0003	0.68 +- 0.1412	0.00 +- 0.0000	0.17 +- 0.0000
			τ	1.45 +- 0.0005	0.52 +- 0.1018	0.30 +- 0.0956	0.29 +- 0.0000
			5τ	3.52 +- 0.0017	0.61 +- 0.1668	1.61 +- 0.3879	0.82 +- 0.0000

WSTL-LD/ST

v	τ	l	p	TOTAL	COMPUTE	STRETCH	IDLE
0.0	50.0μs	0.5τ	0.0	0.10 +- 0.0000	0.06 +- 0.0003	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.16 +- 0.0001	0.11 +- 0.0003	0.00 +- 0.0000	0.01 +- 0.0000
	500.0μs			0.61 +- 0.0000	0.56 +- 0.0003	0.00 +- 0.0000	0.01 +- 0.0000
0.5κ	50.0μs	0.5τ	0.0	0.10 +- 0.0001	0.05 +- 0.0053	0.00 +- 0.0000	0.02 +- 0.0000
	100.0μs			0.22 +- 0.0000	0.13 +- 0.0138	0.00 +- 0.0000	0.04 +- 0.0000
	500.0μs			1.12 +- 0.0009	0.68 +- 0.1587	0.00 +- 0.0000	0.13 +- 0.0000
κ	50.0μs	0.5τ	0.0	0.16 +- 0.0001	0.08 +- 0.0054	0.00 +- 0.0000	0.02 +- 0.0000
	100.0μs			0.22 +- 0.0000	0.13 +- 0.0232	0.00 +- 0.0000	0.04 +- 0.0000
	500.0μs			0.97 +- 0.0003	0.81 +- 0.0388	0.00 +- 0.0000	0.12 +- 0.0000
0.5κ	500.0μs	0.0	0.0	0.71 +- 0.0002	0.63 +- 0.0053	0.00 +- 0.0000	0.06 +- 0.0000
		0.5τ		1.12 +- 0.0009	0.68 +- 0.1587	0.00 +- 0.0000	0.13 +- 0.0000
		τ		0.83 +- 0.0003	0.65 +- 0.0735	0.00 +- 0.0000	0.08 +- 0.0000
0.5κ	500.0μs	0.5τ	0.0	1.12 +- 0.0009	0.68 +- 0.1587	0.00 +- 0.0000	0.13 +- 0.0000
			τ	1.31 +- 0.0006	0.52 +- 0.1142	0.15 +- 0.0442	0.11 +- 0.0000
			5τ	1.82 +- 0.0003	0.61 +- 0.1688	0.75 +- 0.1176	0.16 +- 0.0000

WSTL-GBL

v	τ	l	p	TOTAL	COMPUTE	STRETCH	IDLE
0.0	50.0μs	0.5τ	0.0	1.47 +- 0.0005	0.06 +- 0.0115	0.00 +- 0.0000	1.24 +- 0.0000
	100.0μs			1.53 +- 0.0007	0.11 +- 0.0211	0.00 +- 0.0000	1.22 +- 0.0000
	500.0μs			2.26 +- 0.0015	0.56 +- 0.0022	0.00 +- 0.0000	1.33 +- 0.0000
0.5κ	50.0μs	0.5τ	0.0	1.44 +- 0.0008	0.05 +- 0.0113	0.00 +- 0.0000	1.21 +- 0.0000
	100.0μs			1.79 +- 0.0007	0.13 +- 0.0253	0.00 +- 0.0000	1.43 +- 0.0000
	500.0μs			1.44 +- 0.0207	0.68 +- 0.0190	0.00 +- 0.0000	0.21 +- 0.0000
κ	50.0μs	0.5τ	0.0	2.20 +- 0.0000	0.08 +- 0.0167	0.00 +- 0.0000	1.84 +- 0.0000
	100.0μs			1.79 +- 0.0007	0.13 +- 0.0259	0.00 +- 0.0000	1.43 +- 0.0000
	500.0μs			3.43 +- 0.0031	0.81 +- 0.0132	0.00 +- 0.0000	2.09 +- 0.0000
0.5κ	500.0μs	0.0	0.0	2.28 +- 0.0014	0.63 +- 0.0037	0.00 +- 0.0000	1.52 +- 0.0000

Table A.6: (continued)

		0.5τ		1.44 +- 0.0207	0.68 +- 0.0190	0.00 +- 0.0000	0.21 +- 0.0000
		τ		3.03 +- 0.0020	0.65 +- 0.0065	0.00 +- 0.0000	1.67 +- 0.0000
0.5κ	500.0μs	0.5τ	0.0	1.44 +- 0.0207	0.68 +- 0.0190	0.00 +- 0.0000	0.21 +- 0.0000
			τ	1.14 +- 0.0191	0.52 +- 0.0160	0.00 +- 0.0013	0.13 +- 0.0000
			5τ	1.29 +- 0.0191	0.61 +- 0.0154	0.01 +- 0.0046	0.19 +- 0.0000

WSTL-GBL-LD/ST

v	τ	l	p	TOTAL	COMPUTE	STRETCH	IDLE
0.0	50.0μs	0.5τ	0.0	3.19 +- 0.0017	0.06 +- 0.0442	0.00 +- 0.0000	1.56 +- 0.0000
	100.0μs			3.20 +- 0.0020	0.11 +- 0.0732	0.00 +- 0.0000	1.52 +- 0.0000
	500.0μs			1.12 +- 0.0006	0.56 +- 0.0621	0.00 +- 0.0000	0.20 +- 0.0000
0.5κ	50.0μs	0.5τ	0.0	3.13 +- 0.0010	0.05 +- 0.0431	0.00 +- 0.0000	1.56 +- 0.0000
	100.0μs			3.74 +- 0.0029	0.13 +- 0.0854	0.00 +- 0.0000	1.80 +- 0.0000
	500.0μs			1.20 +- 0.0039	0.68 +- 0.1712	0.00 +- 0.0000	0.04 +- 0.0000
κ	50.0μs	0.5τ	0.0	4.61 +- 0.0028	0.08 +- 0.0651	0.00 +- 0.0000	2.23 +- 0.0000
	100.0μs			3.74 +- 0.0037	0.13 +- 0.0888	0.00 +- 0.0000	1.80 +- 0.0000
	500.0μs			1.72 +- 0.0010	0.81 +- 0.1001	0.00 +- 0.0000	0.35 +- 0.0000
0.5κ	500.0μs	0.0	0.0	2.68 +- 0.0017	0.63 +- 0.2168	0.00 +- 0.0000	1.08 +- 0.0000
		0.5τ		1.20 +- 0.0039	0.68 +- 0.1712	0.00 +- 0.0000	0.04 +- 0.0000
		τ		1.49 +- 0.0008	0.65 +- 0.0547	0.00 +- 0.0000	0.20 +- 0.0000
0.5κ	500.0μs	0.5τ	0.0	1.20 +- 0.0039	0.68 +- 0.1712	0.00 +- 0.0000	0.04 +- 0.0000
			τ	0.92 +- 0.0024	0.52 +- 0.1431	0.00 +- 0.0010	0.04 +- 0.0000
			5τ	1.07 +- 0.0034	0.61 +- 0.1643	0.01 +- 0.0043	0.05 +- 0.0000

Table A.7: Execution times on the SGI Origin 2000 (in seconds) for various task microkernel configurations.

DATA

v	τ	l	p	TOTAL	COMPUTE	STRETCH	IDLE
0.0	50.0μs	0.5τ	0.0	0.06 +- 0.0010	0.06 +- 0.0000	0.00 +- 0.0000	0.00 +- 0.0000
	100.0μs			0.12 +- 0.0010	0.11 +- 0.0001	0.00 +- 0.0000	0.00 +- 0.0000
	500.0μs			0.58 +- 0.0034	0.56 +- 0.0004	0.00 +- 0.0000	0.02 +- 0.0000
0.5κ	50.0μs	0.5τ	0.0	0.10 +- 0.0013	0.05 +- 0.0235	0.00 +- 0.0000	0.04 +- 0.0000
	100.0μs			0.24 +- 0.0013	0.13 +- 0.0540	0.00 +- 0.0000	0.11 +- 0.0000
	500.0μs			1.98 +- 0.3990	0.68 +- 0.3026	0.00 +- 0.0000	1.31 +- 0.0000
κ	50.0μs	0.5τ	0.0	0.19 +- 0.0013	0.08 +- 0.0464	0.00 +- 0.0000	0.10 +- 0.0000
	100.0μs			0.24 +- 0.0030	0.13 +- 0.1060	0.00 +- 0.0000	0.11 +- 0.0000
	500.0μs			1.81 +- 0.0085	0.81 +- 0.4639	0.00 +- 0.0000	1.02 +- 0.0000
0.5κ	500.0μs	0.0	0.0	1.20 +- 0.0058	0.63 +- 0.2233	0.00 +- 0.0000	0.57 +- 0.0000
		0.5τ		1.98 +- 0.3990	0.68 +- 0.3026	0.00 +- 0.0000	1.31 +- 0.0000
		τ		1.20 +- 0.0048	0.65 +- 0.2700	0.00 +- 0.0000	0.56 +- 0.0000

Table A.7: (continued)

0.5κ	500.0μs	0.5τ	0.0	1.98 +- 0.3990	0.68 +- 0.3026	0.00 +- 0.0000	1.31 +- 0.0000
			τ	1.66 +- 0.0082	0.52 +- 0.2690	0.21 +- 0.1943	0.96 +- 0.0000
			5τ	4.45 +- 0.0216	0.68 +- 0.3026	1.42 +- 1.0305	2.60 +- 0.0000

WSHR

v	τ	l	p	TOTAL	COMPUTE	STRETCH	IDLE
0.0	50.0μs	0.5τ	0.0	0.22 +- 0.0011	0.06 +- 0.0001	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.28 +- 0.0015	0.11 +- 0.0003	0.00 +- 0.0000	0.01 +- 0.0000
	500.0μs			0.85 +- 0.0030	0.56 +- 0.0006	0.00 +- 0.0000	0.02 +- 0.0000
0.5κ	50.0μs	0.5τ	0.0	0.20 +- 0.0012	0.05 +- 0.0001	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.32 +- 0.0021	0.13 +- 0.0002	0.00 +- 0.0000	0.01 +- 0.0000
	500.0μs			1.05 +- 0.0052	0.68 +- 0.0009	0.00 +- 0.0000	0.03 +- 0.0000
κ	50.0μs	0.5τ	0.0	0.32 +- 0.0022	0.08 +- 0.0002	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.32 +- 0.0019	0.13 +- 0.0003	0.00 +- 0.0000	0.01 +- 0.0000
	500.0μs			1.23 +- 0.0059	0.81 +- 0.0013	0.00 +- 0.0000	0.04 +- 0.0000
0.5κ	500.0μs	0.0	0.0	0.70 +- 0.0032	0.63 +- 0.0010	0.00 +- 0.0000	0.02 +- 0.0000
		0.5τ		1.05 +- 0.0052	0.68 +- 0.0009	0.00 +- 0.0000	0.03 +- 0.0000
		τ		1.29 +- 0.0123	0.65 +- 0.0011	0.00 +- 0.0000	0.07 +- 0.0000
0.5κ	500.0μs	0.5τ	0.0	1.05 +- 0.0052	0.68 +- 0.0009	0.00 +- 0.0000	0.03 +- 0.0000
			τ	1.09 +- 0.0056	0.52 +- 0.0015	0.22 +- 0.0325	0.10 +- 0.0000
			5τ	3.12 +- 0.0162	0.68 +- 0.0006	1.47 +- 0.2707	0.68 +- 0.0000

WSHR-LOC

v	τ	l	p	TOTAL	COMPUTE	STRETCH	IDLE
0.0	50.0μs	0.5τ	0.0	0.18 +- 0.0025	0.06 +- 0.0012	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.24 +- 0.0019	0.11 +- 0.0034	0.00 +- 0.0000	0.02 +- 0.0000
	500.0μs			0.69 +- 0.0034	0.56 +- 0.0135	0.00 +- 0.0000	0.05 +- 0.0000
0.5κ	50.0μs	0.5τ	0.0	0.18 +- 0.0009	0.05 +- 0.0010	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.27 +- 0.0016	0.13 +- 0.0029	0.00 +- 0.0000	0.01 +- 0.0000
	500.0μs			0.82 +- 0.0402	0.68 +- 0.0149	0.00 +- 0.0000	0.06 +- 0.0000
κ	50.0μs	0.5τ	0.0	0.27 +- 0.0023	0.08 +- 0.0020	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.27 +- 0.0020	0.13 +- 0.0014	0.00 +- 0.0000	0.01 +- 0.0000
	500.0μs			0.94 +- 0.0049	0.81 +- 0.0086	0.00 +- 0.0000	0.03 +- 0.0000
0.5κ	500.0μs	0.0	0.0	0.71 +- 0.0033	0.63 +- 0.0098	0.00 +- 0.0000	0.03 +- 0.0000
		0.5τ		0.82 +- 0.0402	0.68 +- 0.0149	0.00 +- 0.0000	0.06 +- 0.0000
		τ		0.82 +- 0.0046	0.65 +- 0.0145	0.00 +- 0.0000	0.04 +- 0.0000
0.5κ	500.0μs	0.5τ	0.0	0.82 +- 0.0402	0.68 +- 0.0149	0.00 +- 0.0000	0.06 +- 0.0000
			τ	0.92 +- 0.0037	0.52 +- 0.0094	0.23 +- 0.0426	0.11 +- 0.0000
			5τ	2.82 +- 0.0205	0.68 +- 0.0171	1.49 +- 0.2546	0.58 +- 0.0000

WSTL

v	τ	l	p	TOTAL	COMPUTE	STRETCH	IDLE
0.0	50.0μs	0.5τ	0.0	0.12 +- 0.0011	0.06 +- 0.0110	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.17 +- 0.0012	0.11 +- 0.0151	0.00 +- 0.0000	0.01 +- 0.0000

Table A.7: (continued)

	500.0μs			0.72 +- 0.0032	0.56 +- 0.0889	0.00 +- 0.0000	0.06 +- 0.0000
0.5κ	50.0μs	0.5τ	0.0	0.13 +- 0.0011	0.05 +- 0.0106	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.24 +- 0.0057	0.13 +- 0.0291	0.00 +- 0.0000	0.03 +- 0.0000
	500.0μs			1.29 +- 0.7678	0.68 +- 0.1760	0.00 +- 0.0000	0.43 +- 0.0000
κ	50.0μs	0.5τ	0.0	0.22 +- 0.0185	0.08 +- 0.0179	0.00 +- 0.0000	0.01 +- 0.0000
	100.0μs			0.24 +- 0.0142	0.13 +- 0.0324	0.00 +- 0.0000	0.03 +- 0.0000
	500.0μs			1.10 +- 0.0089	0.81 +- 0.1201	0.00 +- 0.0000	0.12 +- 0.0000
0.5κ	500.0μs	0.0	0.0	0.73 +- 0.0075	0.63 +- 0.0593	0.00 +- 0.0000	0.06 +- 0.0000
		0.5τ		1.29 +- 0.7678	0.68 +- 0.1760	0.00 +- 0.0000	0.43 +- 0.0000
		τ		0.99 +- 0.0069	0.65 +- 0.1310	0.00 +- 0.0000	0.11 +- 0.0000
0.5κ	500.0μs	0.5τ	0.0	1.29 +- 0.7678	0.68 +- 0.1760	0.00 +- 0.0000	0.43 +- 0.0000
			τ	1.02 +- 0.0046	0.52 +- 0.0721	0.22 +- 0.1095	0.13 +- 0.0000
			5τ	2.59 +- 0.5695	0.68 +- 0.2066	1.18 +- 0.4058	0.57 +- 0.0000

WSTL-LD/ST

v	τ	l	p	TOTAL	COMPUTE	STRETCH	IDLE
0.0	50.0μs	0.5τ	0.0	0.08 +- 0.0007	0.06 +- 0.0018	0.00 +- 0.0000	0.00 +- 0.0000
	100.0μs			0.13 +- 0.0010	0.11 +- 0.0015	0.00 +- 0.0000	0.00 +- 0.0000
	500.0μs			0.58 +- 0.0099	0.56 +- 0.0098	0.00 +- 0.0000	0.00 +- 0.0000
0.5κ	50.0μs	0.5τ	0.0	0.10 +- 0.0008	0.05 +- 0.0068	0.00 +- 0.0000	0.00 +- 0.0000
	100.0μs			0.18 +- 0.0014	0.13 +- 0.0132	0.00 +- 0.0000	0.00 +- 0.0000
	500.0μs			1.02 +- 0.4491	0.68 +- 0.1904	0.00 +- 0.0000	0.00 +- 0.0000
κ	50.0μs	0.5τ	0.0	0.16 +- 0.0259	0.08 +- 0.0198	0.00 +- 0.0000	0.00 +- 0.0000
	100.0μs			0.18 +- 0.0035	0.13 +- 0.0205	0.00 +- 0.0000	0.00 +- 0.0000
	500.0μs			0.93 +- 0.0210	0.81 +- 0.0365	0.00 +- 0.0000	0.00 +- 0.0000
0.5κ	500.0μs	0.0	0.0	0.67 +- 0.0056	0.63 +- 0.0078	0.00 +- 0.0000	0.00 +- 0.0000
		0.5τ		1.02 +- 0.4491	0.68 +- 0.1904	0.00 +- 0.0000	0.00 +- 0.0000
		τ		0.78 +- 0.0040	0.65 +- 0.0706	0.00 +- 0.0000	0.00 +- 0.0000
0.5κ	500.0μs	0.5τ	0.0	1.02 +- 0.4491	0.68 +- 0.1904	0.00 +- 0.0000	0.00 +- 0.0000
			τ	0.87 +- 0.0039	0.52 +- 0.0603	0.23 +- 0.0833	0.00 +- 0.0000
			5τ	2.16 +- 0.3560	0.68 +- 0.2262	1.25 +- 0.3556	0.00 +- 0.0000

WSTL-GBL

v	τ	l	p	TOTAL	COMPUTE	STRETCH	IDLE
0.0	50.0μs	0.5τ	0.0	1.13 +- 0.0181	0.06 +- 0.0100	0.00 +- 0.0000	0.66 +- 0.0000
	100.0μs			1.11 +- 0.0248	0.11 +- 0.0199	0.00 +- 0.0000	0.59 +- 0.0000
	500.0μs			1.67 +- 0.0166	0.56 +- 0.0113	0.00 +- 0.0000	0.66 +- 0.0000
0.5κ	50.0μs	0.5τ	0.0	1.07 +- 0.0177	0.05 +- 0.0097	0.00 +- 0.0000	0.60 +- 0.0000
	100.0μs			1.26 +- 0.0228	0.13 +- 0.0233	0.00 +- 0.0000	0.65 +- 0.0000
	500.0μs			1.52 +- 0.0363	0.68 +- 0.0272	0.00 +- 0.0000	0.42 +- 0.0000
κ	50.0μs	0.5τ	0.0	1.71 +- 0.0383	0.08 +- 0.0145	0.00 +- 0.0000	1.00 +- 0.0000
	100.0μs			1.26 +- 0.0249	0.13 +- 0.0236	0.00 +- 0.0000	0.65 +- 0.0000
	500.0μs			2.48 +- 0.0317	0.81 +- 0.0192	0.00 +- 0.0000	1.00 +- 0.0000
0.5κ	500.0μs	0.0	0.0	1.66 +- 0.0163	0.63 +- 0.0136	0.00 +- 0.0000	0.75 +- 0.0000

Table A.7: (continued)

		0.5τ		1.52 +- 0.0363	0.68 +- 0.0272	0.00 +- 0.0000	0.42 +- 0.0000
		τ		2.29 +- 0.0264	0.65 +- 0.0045	0.00 +- 0.0000	0.88 +- 0.0000
0.5κ	500.0μs	0.5τ	0.0	1.52 +- 0.0363	0.68 +- 0.0272	0.00 +- 0.0000	0.42 +- 0.0000
			τ	1.18 +- 0.0285	0.52 +- 0.0216	0.03 +- 0.0067	0.30 +- 0.0000
			5τ	1.53 +- 0.0364	0.68 +- 0.0201	0.09 +- 0.0395	0.43 +- 0.0000

WSTL-GBL-LD/ST

v	τ	l	p	TOTAL	COMPUTE	STRETCH	IDLE
0.0	50.0μs	0.5τ	0.0	1.48 +- 0.0063	0.06 +- 0.0409	0.00 +- 0.0000	0.68 +- 0.0000
	100.0μs			1.35 +- 0.0069	0.11 +- 0.1042	0.00 +- 0.0000	0.70 +- 0.0000
	500.0μs			1.36 +- 0.0076	0.56 +- 0.2705	0.00 +- 0.0000	0.28 +- 0.0000
0.5κ	50.0μs	0.5τ	0.0	1.36 +- 0.0083	0.05 +- 0.0396	0.00 +- 0.0000	0.65 +- 0.0000
	100.0μs			1.60 +- 0.0074	0.13 +- 0.0837	0.00 +- 0.0000	0.67 +- 0.0000
	500.0μs			1.36 +- 0.0533	0.68 +- 0.1165	0.00 +- 0.0000	0.00 +- 0.0000
κ	50.0μs	0.5τ	0.0	1.93 +- 0.0144	0.08 +- 0.0720	0.00 +- 0.0000	0.97 +- 0.0000
	100.0μs			1.60 +- 0.0135	0.13 +- 0.1465	0.00 +- 0.0000	0.67 +- 0.0000
	500.0μs			2.22 +- 0.0157	0.81 +- 0.5480	0.00 +- 0.0000	0.75 +- 0.0000
0.5κ	500.0μs	0.0	0.0	1.34 +- 0.0084	0.63 +- 0.3086	0.00 +- 0.0000	0.27 +- 0.0000
		0.5τ		1.36 +- 0.0533	0.68 +- 0.1165	0.00 +- 0.0000	0.00 +- 0.0000
		τ		1.96 +- 0.0100	0.65 +- 0.3771	0.00 +- 0.0000	0.52 +- 0.0000
0.5κ	500.0μs	0.5τ	0.0	1.36 +- 0.0533	0.68 +- 0.1165	0.00 +- 0.0000	0.00 +- 0.0000
			τ	1.06 +- 0.0096	0.52 +- 0.2178	0.02 +- 0.0487	0.12 +- 0.0000
			5τ	1.37 +- 0.0379	0.68 +- 0.5136	0.08 +- 0.3759	0.12 +- 0.0000

APPENDIX B

RAW APPLICATION DATA

The application results presented in Chapter 7 were in the form of speedup plots and normalized execution time breakdowns. For completeness, this appendix contains tables listing the raw execution data for each of the applications.

B.1 IC-Cedar

Tables B.1 and B.2 list the execution times for the IC-Cedar application on the Cray T3D and the SGI Origin 2000, respectively.

Measurements are presented for different numbers of processors, across the range of communication architectures, and for different levels of support of data locality and load balancing. The prefix of the configuration label denotes support for data locality and load balance: BASE, BASE+DATA, BASE+VC, BASE+VC+DATA correspond to the baseline version of the application, and its augmentation with various load balancing policies and custom view coherence protocols described in Section 7.2. The TOTAL, COMPUTE, IDLE, COHERENCE, and OVERHEAD columns show the total execution time, and its breakdown into compute time, idle time, coherence time, and runtime overhead components. The data in the tables are displayed graphically in Figures 7.6–7.8.

Table B.1: IC-Cedar execution times (in seconds) on the Cray T3D.

BASE-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	265.67 +- 0.000	117.03 +- 0.000	0.00 +- 0.000	81.31 +- 0.000	67.32 +- 0.000
2	290.39 +- 0.000	61.64 +- 0.265	122.95 +- 1.517	70.94 +- 0.108	34.86 +- 1.360
4	242.66 +- 0.000	30.73 +- 0.458	141.51 +- 1.209	61.75 +- 0.211	8.68 +- 0.664
8	183.69 +- 0.000	15.83 +- 0.225	112.92 +- 0.787	57.71 +- 0.135	5.77 +- 0.534

Table B.1: (continued)

16	139.82 +- 0.000	8.34 +- 0.176	83.33 +- 0.937	56.40 +- 0.189	4.25 +- 0.828
32	116.63 +- 0.000	4.56 +- 0.139	68.93 +- 1.371	53.55 +- 0.394	3.41 +- 1.174
64	89.66 +- 0.000	2.55 +- 0.098	51.99 +- 1.645	43.00 +- 0.731	2.88 +- 1.482

BASE-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	303.01 +- 0.000	106.08 +- 0.000	0.00 +- 0.000	81.31 +- 0.000	115.61 +- 0.000
2	301.77 +- 0.000	54.88 +- 0.105	129.89 +- 1.656	54.24 +- 0.109	62.76 +- 1.660
4	247.59 +- 0.000	27.77 +- 0.363	149.05 +- 1.480	39.87 +- 0.212	30.90 +- 1.046
8	177.85 +- 0.000	14.29 +- 0.207	113.48 +- 0.685	33.26 +- 0.125	16.83 +- 0.487
16	128.83 +- 0.000	7.54 +- 0.177	80.08 +- 0.934	30.87 +- 0.144	10.34 +- 0.841
32	99.96 +- 0.000	4.11 +- 0.129	60.08 +- 1.251	29.25 +- 0.253	6.53 +- 1.122
64	75.01 +- 0.000	2.29 +- 0.088	43.48 +- 1.604	23.42 +- 0.457	5.82 +- 1.523

BASE-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	326.77 +- 0.000	119.09 +- 0.000	0.00 +- 0.000	162.58 +- 0.000	45.11 +- 0.000
2	374.35 +- 0.000	62.85 +- 0.201	178.43 +- 0.651	104.46 +- 0.217	28.60 +- 1.069
4	210.81 +- 0.000	31.17 +- 0.700	98.45 +- 3.292	69.91 +- 0.422	11.27 +- 2.306
8	133.59 +- 0.000	16.02 +- 0.361	59.12 +- 1.926	53.21 +- 0.238	5.25 +- 1.482
16	102.36 +- 0.000	8.53 +- 0.233	44.15 +- 1.823	45.69 +- 0.272	3.99 +- 1.536
32	91.68 +- 0.000	4.76 +- 0.164	41.03 +- 2.496	40.87 +- 0.325	5.02 +- 2.349
64	186.25 +- 0.000	2.72 +- 0.111	144.18 +- 14.087	31.94 +- 0.480	7.42 +- 14.106

BASE+DATA-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	267.18 +- 0.000	117.01 +- 0.000	0.00 +- 0.000	81.31 +- 0.000	68.86 +- 0.000
2	293.82 +- 0.000	60.37 +- 1.607	136.71 +- 2.307	61.69 +- 0.138	35.04 +- 0.561
4	185.56 +- 0.000	29.84 +- 3.389	99.15 +- 3.477	41.60 +- 1.173	14.97 +- 1.326
8	107.85 +- 0.000	14.80 +- 2.345	59.65 +- 1.720	27.49 +- 0.483	5.91 +- 1.258
16	68.67 +- 0.000	7.41 +- 2.172	41.31 +- 1.472	17.73 +- 1.251	2.22 +- 1.544
32	43.09 +- 0.000	3.67 +- 1.409	27.40 +- 1.457	11.29 +- 1.290	0.73 +- 1.201
64	28.77 +- 0.000	1.84 +- 0.822	19.44 +- 1.038	7.21 +- 0.939	0.27 +- 0.866

BASE+DATA-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	303.01 +- 0.000	106.08 +- 0.000	0.00 +- 0.000	81.31 +- 0.000	115.61 +- 0.000
2	301.77 +- 0.000	54.88 +- 0.105	129.89 +- 1.656	54.24 +- 0.109	62.76 +- 1.660
4	247.59 +- 0.000	27.77 +- 0.363	149.05 +- 1.480	39.87 +- 0.212	30.90 +- 1.046
8	177.85 +- 0.000	14.29 +- 0.207	113.48 +- 0.685	33.26 +- 0.125	16.83 +- 0.487
16	128.83 +- 0.000	7.54 +- 0.177	80.08 +- 0.934	30.87 +- 0.144	10.34 +- 0.841
32	99.96 +- 0.000	4.11 +- 0.129	60.08 +- 1.251	29.25 +- 0.253	6.53 +- 1.122
64	75.01 +- 0.000	2.29 +- 0.088	43.48 +- 1.604	23.42 +- 0.457	5.82 +- 1.523

Table B.1: (continued)

BASE+DATA-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	323.18 +- 0.000	117.22 +- 0.000	0.00 +- 0.000	162.58 +- 0.000	43.37 +- 0.000
2	177.17 +- 0.000	58.14 +- 0.854	1.39 +- 1.059	97.26 +- 0.214	20.38 +- 0.009
4	98.70 +- 0.000	28.95 +- 3.187	3.48 +- 2.449	55.81 +- 1.151	10.46 +- 0.860
8	55.53 +- 0.000	14.57 +- 2.310	3.46 +- 1.855	32.40 +- 0.942	5.10 +- 1.048
16	32.44 +- 0.000	7.30 +- 2.172	3.59 +- 1.296	18.85 +- 0.799	2.71 +- 1.154
32	19.47 +- 0.000	3.64 +- 1.406	3.34 +- 1.317	11.02 +- 0.771	1.47 +- 0.832
64	14.58 +- 0.000	1.83 +- 0.825	5.22 +- 1.041	6.55 +- 0.607	0.97 +- 0.787

BASE+VC-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	284.04 +- 0.000	117.80 +- 0.000	0.00 +- 0.000	78.79 +- 0.000	87.45 +- 0.000
2	272.86 +- 0.000	61.26 +- 0.256	121.39 +- 0.354	55.01 +- 0.109	35.20 +- 0.720
4	225.59 +- 0.000	30.72 +- 0.321	139.80 +- 0.765	43.14 +- 0.212	11.93 +- 0.377
8	144.23 +- 0.000	15.84 +- 0.212	89.17 +- 0.343	37.20 +- 0.123	2.02 +- 0.097
16	88.45 +- 0.000	8.36 +- 0.170	48.69 +- 0.367	34.03 +- 0.143	1.84 +- 0.176
32	60.17 +- 0.000	4.58 +- 0.137	29.70 +- 0.447	30.45 +- 0.233	1.56 +- 0.238
64	49.18 +- 0.000	2.57 +- 0.098	27.09 +- 0.520	23.88 +- 0.398	1.37 +- 0.201

BASE+VC-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	291.43 +- 0.000	103.26 +- 0.000	0.00 +- 0.000	78.79 +- 0.000	109.38 +- 0.000
2	152.60 +- 0.000	54.12 +- 0.142	1.24 +- 1.059	43.72 +- 0.109	53.53 +- 1.026
4	82.95 +- 0.000	27.52 +- 0.502	2.35 +- 1.299	26.19 +- 0.213	26.89 +- 0.655
8	49.57 +- 0.000	14.16 +- 0.250	1.73 +- 0.592	17.45 +- 0.118	16.23 +- 0.341
16	32.63 +- 0.000	7.51 +- 0.188	1.78 +- 0.458	13.05 +- 0.109	10.29 +- 0.261
32	23.05 +- 0.000	4.15 +- 0.133	1.67 +- 0.421	10.33 +- 0.140	6.90 +- 0.270
64	17.33 +- 0.000	2.34 +- 0.090	2.55 +- 0.530	7.61 +- 0.229	4.82 +- 0.321

BASE+VC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	291.44 +- 0.000	103.25 +- 0.000	0.00 +- 0.000	78.79 +- 0.000	109.39 +- 0.000
2	152.61 +- 0.000	54.12 +- 0.155	1.28 +- 1.100	47.56 +- 0.110	49.66 +- 1.055
4	82.98 +- 0.000	27.52 +- 0.505	2.38 +- 1.323	31.95 +- 0.213	21.14 +- 0.672
8	49.57 +- 0.000	14.16 +- 0.250	1.73 +- 0.596	24.16 +- 0.118	9.52 +- 0.343
16	32.66 +- 0.000	7.51 +- 0.188	1.80 +- 0.456	20.18 +- 0.120	3.17 +- 0.255
32	23.04 +- 0.000	4.15 +- 0.133	1.67 +- 0.432	17.17 +- 0.206	0.60 +- 0.244
64	17.24 +- 0.000	2.34 +- 0.090	2.46 +- 0.541	13.14 +- 0.385	0.49 +- 0.244

Table B.1: (continued)

BASE+VC+DATA-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	284.36 +- 0.000	117.82 +- 0.000	0.00 +- 0.000	78.79 +- 0.000	87.75 +- 0.000
2	282.64 +- 0.000	60.10 +- 1.125	135.14 +- 1.534	50.20 +- 0.078	37.20 +- 0.332
4	171.56 +- 0.000	29.87 +- 3.363	93.88 +- 3.173	31.20 +- 0.959	16.61 +- 0.935
8	86.62 +- 0.000	14.82 +- 2.327	45.50 +- 1.847	19.51 +- 0.667	6.79 +- 1.144
16	48.83 +- 0.000	7.43 +- 2.179	26.57 +- 1.368	12.07 +- 0.581	2.76 +- 1.264
32	27.29 +- 0.000	3.68 +- 1.415	15.11 +- 1.173	7.44 +- 0.605	1.06 +- 0.906
64	16.42 +- 0.000	1.85 +- 0.827	9.47 +- 0.743	4.64 +- 0.481	0.47 +- 0.604

BASE+VC+DATA-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	290.00 +- 0.000	103.39 +- 0.000	0.00 +- 0.000	78.79 +- 0.000	107.82 +- 0.000
2	150.59 +- 0.000	52.87 +- 1.411	2.00 +- 1.930	42.32 +- 0.059	53.40 +- 0.460
4	76.53 +- 0.000	26.17 +- 3.051	1.41 +- 1.608	22.69 +- 0.769	26.25 +- 2.326
8	42.17 +- 0.000	13.19 +- 2.116	2.12 +- 1.837	12.39 +- 0.760	14.47 +- 2.366
16	23.99 +- 0.000	6.59 +- 1.942	2.94 +- 1.166	6.81 +- 0.525	7.65 +- 2.259
32	13.44 +- 0.000	3.27 +- 1.259	2.45 +- 0.932	3.77 +- 0.365	3.97 +- 1.386
64	10.72 +- 0.000	1.64 +- 0.739	4.58 +- 0.813	2.13 +- 0.259	2.38 +- 1.028

BASE+VC+DATA-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	290.01 +- 0.000	103.40 +- 0.000	0.00 +- 0.000	78.79 +- 0.000	107.82 +- 0.000
2	150.60 +- 0.000	52.87 +- 1.407	2.00 +- 1.927	44.99 +- 0.030	50.73 +- 0.490
4	76.53 +- 0.000	26.17 +- 3.054	1.43 +- 1.594	25.58 +- 1.112	23.35 +- 2.276
8	42.18 +- 0.000	13.19 +- 2.117	2.09 +- 1.806	14.81 +- 1.015	12.09 +- 2.259
16	23.96 +- 0.000	6.60 +- 1.942	2.91 +- 1.178	8.59 +- 0.688	5.86 +- 2.221
32	13.41 +- 0.000	3.27 +- 1.260	2.42 +- 0.923	5.02 +- 0.492	2.71 +- 1.371
64	10.53 +- 0.000	1.64 +- 0.739	4.39 +- 0.798	2.98 +- 0.384	1.52 +- 0.990

Table B.2: IC-Cedar execution times (in seconds) on the SGI Origin 2000.

BASE-CC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	22.17 +- 0.000	22.17 +- 0.000	0.00 +- 0.000	0.00 +- 0.000	0.00 +- 0.000
2	11.61 +- 0.003	11.49 +- 0.019	0.12 +- 0.016	0.00 +- 0.000	0.00 +- 0.000
4	6.27 +- 0.003	6.00 +- 0.131	0.26 +- 0.132	0.00 +- 0.000	0.00 +- 0.001
8	3.50 +- 0.008	3.28 +- 0.139	0.21 +- 0.138	0.00 +- 0.000	0.01 +- 0.003
16	1.97 +- 0.028	1.77 +- 0.089	0.19 +- 0.099	0.00 +- 0.000	0.01 +- 0.005

Table B.2: (continued)

BASE-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	35.31 +- 0.000	19.77 +- 0.000	0.00 +- 0.000	15.25 +- 0.000	0.30 +- 0.000
2	46.49 +- 0.011	6.11 +- 0.861	17.23 +- 0.871	16.86 +- 0.020	6.29 +- 0.000
4	44.28 +- 0.045	-2.57 +- 0.739	20.56 +- 0.690	16.95 +- 0.040	9.33 +- 0.004
8	57.78 +- 0.136	4.84 +- 0.959	25.00 +- 0.973	17.10 +- 0.029	10.84 +- 0.014
16	63.25 +- 0.144	6.61 +- 1.433	27.80 +- 1.378	17.27 +- 0.049	11.57 +- 0.031

BASE-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	32.44 +- 0.000	16.89 +- 0.000	0.00 +- 0.000	15.25 +- 0.000	0.30 +- 0.000
2	38.53 +- 0.012	8.50 +- 1.053	14.98 +- 1.021	9.73 +- 0.020	5.32 +- 0.000
4	42.91 +- 0.015	8.95 +- 1.082	19.60 +- 1.095	6.49 +- 0.040	7.87 +- 0.003
8	46.22 +- 0.087	12.04 +- 0.592	20.05 +- 0.557	4.99 +- 0.023	9.14 +- 0.009
16	56.25 +- 0.103	17.15 +- 0.517	24.93 +- 0.494	4.44 +- 0.025	9.73 +- 0.020

BASE-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	46.02 +- 0.000	20.32 +- 0.000	0.00 +- 0.000	25.40 +- 0.000	0.30 +- 0.000
2	32.89 +- 0.005	10.18 +- 0.032	0.04 +- 0.007	20.25 +- 0.034	2.40 +- 0.000
4	23.62 +- 0.055	4.08 +- 0.124	0.34 +- 0.179	15.73 +- 0.066	3.47 +- 0.001
8	29.26 +- 0.075	8.33 +- 2.484	3.36 +- 2.517	13.58 +- 0.038	4.00 +- 0.006
16	28.29 +- 0.473	7.52 +- 1.833	3.88 +- 1.836	12.64 +- 0.052	4.25 +- 0.017

BASE+DATA-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	32.35 +- 0.000	16.81 +- 0.000	0.00 +- 0.000	15.25 +- 0.000	0.30 +- 0.000
2	41.58 +- 0.160	4.87 +- 1.113	17.05 +- 0.624	14.05 +- 0.029	5.60 +- 0.299
4	29.12 +- 0.160	0.74 +- 1.031	13.16 +- 0.811	10.49 +- 0.364	4.72 +- 0.280
8	19.73 +- 0.046	-0.33 +- 0.550	8.72 +- 0.526	7.41 +- 0.119	3.92 +- 0.074
16	15.58 +- 0.059	-0.14 +- 0.555	7.82 +- 0.635	5.00 +- 0.425	2.90 +- 0.296

BASE+DATA-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	33.80 +- 0.000	18.25 +- 0.000	0.00 +- 0.000	15.25 +- 0.000	0.30 +- 0.000
2	46.92 +- 0.393	13.85 +- 1.156	19.07 +- 1.087	9.07 +- 0.022	4.93 +- 0.303
4	40.05 +- 1.238	8.77 +- 0.940	22.09 +- 2.121	5.19 +- 0.089	4.00 +- 0.227
8	20.55 +- 0.028	5.43 +- 0.365	8.79 +- 0.358	3.01 +- 0.073	3.31 +- 0.066
16	13.86 +- 0.062	2.97 +- 0.259	6.69 +- 0.281	1.75 +- 0.081	2.44 +- 0.269

Table B.2: (continued)

BASE+DATA-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	46.22 +- 0.000	20.51 +- 0.000	0.00 +- 0.000	25.40 +- 0.000	0.30 +- 0.000
2	59.36 +- 11.819	20.40 +- 1.036	18.16 +- 12.524	17.94 +- 0.038	2.87 +- 0.294
4	37.74 +- 0.694	16.21 +- 0.414	8.39 +- 0.543	11.34 +- 0.323	1.80 +- 0.085
8	25.29 +- 0.044	11.18 +- 4.799	5.53 +- 4.802	7.11 +- 0.183	1.47 +- 0.028
16	8.51 +- 0.079	1.18 +- 0.543	1.84 +- 0.584	4.41 +- 0.223	1.09 +- 0.117

BASE+VC-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	32.31 +- 0.000	17.54 +- 0.000	0.00 +- 0.000	14.77 +- 0.000	0.00 +- 0.000
2	34.54 +- 0.007	4.96 +- 0.855	14.55 +- 0.868	12.11 +- 0.020	2.92 +- 0.000
4	38.84 +- 0.031	3.16 +- 0.923	20.50 +- 0.945	10.78 +- 0.040	4.40 +- 0.003
8	28.50 +- 0.062	-0.01 +- 0.400	13.26 +- 0.403	10.12 +- 0.025	5.13 +- 0.006
16	29.46 +- 0.138	3.22 +- 0.539	11.04 +- 0.581	9.72 +- 0.033	5.48 +- 0.022

BASE+VC-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	31.75 +- 0.000	16.98 +- 0.000	0.00 +- 0.000	14.77 +- 0.000	0.00 +- 0.000
2	18.05 +- 0.006	8.85 +- 0.024	0.10 +- 0.051	8.12 +- 0.021	0.97 +- 0.000
4	12.95 +- 0.008	6.07 +- 0.462	0.61 +- 0.455	4.80 +- 0.040	1.47 +- 0.001
8	10.34 +- 0.052	4.78 +- 0.351	0.70 +- 0.344	3.14 +- 0.022	1.72 +- 0.006
16	9.10 +- 0.391	3.96 +- 0.353	1.00 +- 0.422	2.31 +- 0.020	1.83 +- 0.008

BASE+VC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	34.43 +- 0.000	19.65 +- 0.000	0.00 +- 0.000	14.77 +- 0.000	0.00 +- 0.000
2	18.05 +- 0.009	7.98 +- 0.022	0.10 +- 0.052	8.99 +- 0.021	0.97 +- 0.000
4	12.91 +- 0.005	4.52 +- 0.442	0.82 +- 0.472	6.10 +- 0.040	1.47 +- 0.001
8	10.71 +- 0.051	3.55 +- 0.418	0.79 +- 0.419	4.66 +- 0.022	1.72 +- 0.007
16	10.09 +- 0.492	3.22 +- 0.336	1.11 +- 0.737	3.92 +- 0.023	1.84 +- 0.009

BASE+VC+DATA-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	35.00 +- 0.000	20.22 +- 0.000	0.00 +- 0.000	14.77 +- 0.000	0.00 +- 0.000
2	37.30 +- 0.181	7.46 +- 0.235	17.13 +- 0.033	10.67 +- 0.019	2.04 +- 0.003
4	21.62 +- 0.023	1.69 +- 0.829	10.51 +- 0.536	7.20 +- 0.222	2.22 +- 0.129
8	12.86 +- 0.009	0.37 +- 0.239	5.83 +- 0.252	4.79 +- 0.109	1.87 +- 0.035
16	9.56 +- 0.032	0.85 +- 0.354	4.21 +- 0.307	3.10 +- 0.203	1.39 +- 0.153

Table B.2: (continued)

BASE+VC+DATA-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	34.98 +- 0.000	20.20 +- 0.000	0.00 +- 0.000	14.77 +- 0.000	0.00 +- 0.000
2	16.91 +- 0.013	8.32 +- 0.010	0.03 +- 0.011	7.88 +- 0.012	0.68 +- 0.003
4	11.21 +- 0.005	5.78 +- 0.129	0.48 +- 0.274	4.20 +- 0.138	0.75 +- 0.039
8	6.27 +- 0.031	2.97 +- 0.149	0.39 +- 0.257	2.28 +- 0.138	0.64 +- 0.021
16	4.00 +- 0.079	1.61 +- 0.145	0.65 +- 0.177	1.24 +- 0.095	0.49 +- 0.068

BASE+VC+DATA-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	35.31 +- 0.000	20.53 +- 0.000	0.00 +- 0.000	14.77 +- 0.000	0.00 +- 0.000
2	18.51 +- 0.002	9.22 +- 0.087	0.13 +- 0.092	8.49 +- 0.005	0.68 +- 0.003
4	10.88 +- 0.021	4.90 +- 0.158	0.38 +- 0.323	4.85 +- 0.215	0.75 +- 0.038
8	6.39 +- 0.015	2.52 +- 0.112	0.42 +- 0.234	2.82 +- 0.195	0.63 +- 0.022
16	4.13 +- 0.088	1.38 +- 0.166	0.63 +- 0.159	1.65 +- 0.132	0.49 +- 0.069

B.2 Radiosity

Tables B.3 and B.4 list the execution times for the radiosity application on the Cray T3D and the SGI Origin 2000, respectively.

Measurements are presented for different numbers of processors, across the range of communication architectures, and for different levels of support of data locality and load balancing. The prefix of the configuration label denotes support for data locality and load balance: BASE, BASE+WSHR, BASE+WSHRLOC, BASE+WSTL, BASE+VC, BASE+VC+WSHR, BASE+VC+WSHRLOC, and BASE+VC+WSTL correspond to the baseline version of the application, and its augmentation with various load balancing policies and custom view coherence protocols described in Section 7.3. All data locality configurations use the `READ_CONSTANT` and `READ_SELF_INV` protocols. When the `READ_SYNC` protocol is used instead, the BASE+NOFS+WSHR configuration shows the corresponding execution times. All load-balancing configurations show the effect of both object placement and thread placement policies. The BASE+VC+DATA configuration shows execution times when only object placement policies are used for load balance. The TOTAL, COMPUTE, IDLE, COHERENCE, and OVERHEAD columns show the total execution time, and its breakdown into compute time, idle time, coherence time, and run-time overhead components. The data in the tables are displayed graphically in Figures 7.10–7.14.

Table B.3: Radiosity execution times (in seconds) on the Cray T3D.

BASE-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	492.26 +- 0.000	378.26 +- 0.000	0.71 +- 0.000	32.13 +- 0.000	81.16 +- 0.000
2	538.54 +- 0.000	203.87 +- 7.924	258.77 +- 8.083	29.45 +- 0.290	46.45 +- 0.130
4	459.80 +- 0.000	101.49 +- 24.528	315.04 +- 29.412	20.45 +- 2.127	22.81 +- 2.893
8	352.04 +- 0.000	51.02 +- 15.387	276.86 +- 17.414	12.85 +- 1.308	11.31 +- 1.974
16	286.53 +- 0.000	25.51 +- 15.368	247.56 +- 19.797	7.88 +- 2.372	5.59 +- 2.701
32	252.95 +- 0.000	12.77 +- 11.480	232.66 +- 16.092	4.61 +- 2.453	2.92 +- 2.608
64	208.57 +- 0.001	6.32 +- 8.752	197.89 +- 12.555	2.61 +- 2.062	1.76 +- 2.062

BASE-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	492.26 +- 0.000	378.26 +- 0.000	0.71 +- 0.000	32.13 +- 0.000	81.16 +- 0.000
2	531.49 +- 0.000	204.10 +- 7.348	256.26 +- 7.553	23.06 +- 0.199	48.07 +- 0.005
4	468.95 +- 0.000	102.48 +- 24.532	327.42 +- 29.100	14.48 +- 1.597	24.58 +- 3.050
8	351.56 +- 0.000	51.15 +- 14.996	279.37 +- 16.823	8.75 +- 1.062	12.30 +- 1.786
16	291.60 +- 0.000	25.64 +- 15.415	254.56 +- 19.506	5.17 +- 1.675	6.22 +- 2.904
32	257.00 +- 0.000	12.63 +- 11.053	238.17 +- 15.171	2.93 +- 1.675	3.26 +- 2.752
64	182.74 +- 0.001	6.31 +- 8.731	172.91 +- 12.087	1.60 +- 1.368	1.93 +- 2.240

BASE-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	492.26 +- 0.000	378.26 +- 0.000	0.71 +- 0.000	32.13 +- 0.000	81.16 +- 0.000
2	302.30 +- 0.000	193.20 +- 5.956	22.50 +- 5.916	35.35 +- 0.217	51.26 +- 0.256
4	224.85 +- 0.000	99.60 +- 22.556	76.66 +- 28.146	21.27 +- 2.333	27.32 +- 3.322
8	133.67 +- 0.000	49.92 +- 14.883	57.77 +- 17.333	12.31 +- 1.365	13.66 +- 1.943
16	109.39 +- 0.000	25.26 +- 15.737	70.32 +- 20.628	7.05 +- 2.284	6.77 +- 3.140
32	77.93 +- 0.000	12.41 +- 10.831	58.00 +- 15.771	4.02 +- 2.354	3.50 +- 2.946
64	84.65 +- 0.000	6.29 +- 8.672	74.12 +- 12.578	2.17 +- 1.917	2.06 +- 2.277

BASE+WSHR-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	510.13 +- 0.000	391.33 +- 0.000	0.00 +- 0.000	32.70 +- 0.000	86.10 +- 0.000
2	472.81 +- 0.000	204.44 +- 0.116	171.80 +- 0.137	37.43 +- 0.001	59.14 +- 0.253
4	301.07 +- 0.000	101.26 +- 0.153	139.34 +- 0.373	28.48 +- 0.114	32.00 +- 0.142
8	166.56 +- 0.000	50.50 +- 0.184	80.43 +- 0.693	19.25 +- 0.271	16.37 +- 0.563
16	89.00 +- 0.000	25.12 +- 0.108	43.46 +- 0.602	12.34 +- 0.177	8.08 +- 0.505
32	48.80 +- 0.000	12.60 +- 0.093	24.78 +- 0.538	7.57 +- 0.269	3.86 +- 0.346
64	28.75 +- 0.001	6.29 +- 0.049	15.97 +- 0.361	4.46 +- 0.388	2.04 +- 0.324

BASE+WSHR-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	511.78 +- 0.000	391.17 +- 0.000	0.00 +- 0.000	32.70 +- 0.000	87.91 +- 0.000

Table B.3: (continued)

2	466.79 +- 0.000	204.20 +- 0.045	170.73 +- 0.028	26.75 +- 0.017	65.11 +- 0.034
4	290.26 +- 0.000	101.36 +- 0.163	134.18 +- 0.354	18.22 +- 0.074	36.50 +- 0.160
8	158.21 +- 0.000	50.42 +- 0.151	76.75 +- 0.808	11.70 +- 0.222	19.34 +- 0.625
16	83.44 +- 0.000	25.30 +- 0.126	41.00 +- 0.567	7.36 +- 0.136	9.78 +- 0.528
32	45.13 +- 0.000	12.62 +- 0.088	23.08 +- 0.545	4.54 +- 0.183	4.89 +- 0.408
64	25.66 +- 0.001	6.32 +- 0.054	14.11 +- 0.358	2.62 +- 0.197	2.62 +- 0.312

BASE+WSHR-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	539.69 +- 0.000	389.60 +- 0.000	0.00 +- 0.000	49.19 +- 0.000	100.91 +- 0.000
2	294.83 +- 0.000	197.08 +- 0.264	0.88 +- 0.348	40.80 +- 0.033	56.07 +- 0.051
4	158.35 +- 0.000	99.27 +- 0.222	1.50 +- 0.414	27.19 +- 0.094	30.39 +- 0.125
8	84.61 +- 0.000	49.86 +- 0.131	2.14 +- 0.642	17.03 +- 0.202	15.59 +- 0.437
16	46.27 +- 0.000	24.93 +- 0.110	2.98 +- 0.445	10.47 +- 0.208	7.89 +- 0.381
32	26.59 +- 0.000	12.55 +- 0.071	3.85 +- 0.465	6.28 +- 0.233	3.90 +- 0.279
64	17.21 +- 0.001	6.26 +- 0.068	5.20 +- 0.383	3.58 +- 0.281	2.16 +- 0.264

BASE+WSHRLOC-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	510.59 +- 0.000	391.45 +- 0.000	0.00 +- 0.000	32.70 +- 0.000	86.44 +- 0.000
2	501.86 +- 0.000	204.99 +- 0.106	201.99 +- 0.715	37.37 +- 0.002	57.51 +- 0.607
4	306.14 +- 0.000	101.04 +- 0.220	143.99 +- 0.454	28.97 +- 0.129	32.14 +- 0.244
8	167.87 +- 0.000	50.56 +- 0.224	81.34 +- 0.654	19.45 +- 0.216	16.52 +- 0.619
16	87.88 +- 0.000	25.09 +- 0.227	42.56 +- 0.585	12.21 +- 0.326	8.02 +- 0.504
32	48.64 +- 0.000	12.62 +- 0.114	24.90 +- 0.577	7.21 +- 0.291	3.90 +- 0.364
64	29.05 +- 0.001	6.30 +- 0.254	16.56 +- 0.494	4.20 +- 0.403	2.00 +- 0.297

BASE+WSHRLOC-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	513.76 +- 0.000	391.46 +- 0.000	0.00 +- 0.000	32.70 +- 0.000	89.60 +- 0.000
2	496.73 +- 0.000	204.66 +- 0.006	202.17 +- 0.555	26.73 +- 0.016	63.18 +- 0.544
4	295.91 +- 0.000	101.29 +- 0.267	139.82 +- 0.540	18.43 +- 0.085	36.37 +- 0.251
8	158.91 +- 0.000	50.43 +- 0.218	77.48 +- 0.882	11.80 +- 0.191	19.20 +- 0.677
16	84.01 +- 0.000	25.31 +- 0.212	41.91 +- 0.711	7.23 +- 0.176	9.57 +- 0.528
32	44.91 +- 0.000	12.64 +- 0.277	23.20 +- 0.575	4.26 +- 0.189	4.81 +- 0.373
64	26.31 +- 0.001	6.33 +- 0.375	15.01 +- 0.443	2.46 +- 0.199	2.51 +- 0.317

BASE+WSHRLOC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	541.15 +- 0.000	389.54 +- 0.000	0.00 +- 0.000	49.19 +- 0.000	102.42 +- 0.000
2	292.85 +- 0.000	196.33 +- 0.002	1.48 +- 0.407	40.76 +- 0.029	54.28 +- 0.381
4	158.51 +- 0.000	99.20 +- 0.164	1.72 +- 0.348	27.51 +- 0.115	30.09 +- 0.170
8	85.20 +- 0.000	49.79 +- 0.219	2.82 +- 0.470	17.01 +- 0.230	15.58 +- 0.458

Table B.3: (continued)

16	46.54 +- 0.000	24.92 +- 0.309	3.47 +- 0.432	10.30 +- 0.222	7.86 +- 0.382
32	27.02 +- 0.000	12.56 +- 0.259	4.58 +- 0.499	5.92 +- 0.240	3.96 +- 0.316
64	17.52 +- 0.001	6.28 +- 0.286	5.68 +- 0.480	3.43 +- 0.289	2.14 +- 0.247

BASE+WSTL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	513.11 +- 0.000	382.28 +- 0.000	18.71 +- 0.000	32.46 +- 0.000	79.66 +- 0.000
2	360.30 +- 0.000	194.85 +- 0.176	74.60 +- 0.303	34.28 +- 0.009	56.56 +- 0.118
4	233.65 +- 0.000	98.11 +- 1.900	80.70 +- 1.739	24.94 +- 0.107	29.89 +- 0.164
8	144.80 +- 0.000	49.34 +- 7.720	63.35 +- 6.933	16.34 +- 0.396	15.76 +- 0.550
16	81.00 +- 0.002	24.61 +- 5.268	37.91 +- 4.477	10.41 +- 0.335	8.08 +- 0.551
32	45.38 +- 0.000	12.33 +- 2.113	22.47 +- 1.597	6.20 +- 0.304	4.39 +- 0.366
64	28.85 +- 0.002	6.19 +- 0.768	16.31 +- 0.668	3.63 +- 0.354	2.72 +- 0.443

BASE+WSTL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	516.76 +- 0.000	384.29 +- 0.000	18.79 +- 0.000	32.46 +- 0.000	81.22 +- 0.000
2	352.73 +- 0.000	195.71 +- 0.965	70.84 +- 1.058	25.34 +- 0.049	60.84 +- 0.044
4	226.94 +- 0.000	98.40 +- 2.579	79.02 +- 2.507	16.54 +- 0.081	32.98 +- 0.081
8	137.13 +- 0.002	49.16 +- 7.133	59.76 +- 6.242	10.34 +- 0.237	17.86 +- 0.727
16	77.09 +- 0.003	24.72 +- 5.924	36.60 +- 5.115	6.41 +- 0.243	9.36 +- 0.623
32	42.48 +- 0.000	12.40 +- 1.909	21.26 +- 1.403	3.76 +- 0.212	5.06 +- 0.412
64	26.39 +- 0.002	6.21 +- 0.683	14.97 +- 0.580	2.18 +- 0.236	3.04 +- 0.363

BASE+WSTL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	546.14 +- 0.000	382.10 +- 0.000	18.59 +- 0.000	48.83 +- 0.000	96.62 +- 0.000
2	290.06 +- 0.000	192.51 +- 0.113	8.84 +- 0.212	38.69 +- 0.046	50.03 +- 0.145
4	154.13 +- 0.000	96.87 +- 0.245	5.45 +- 0.339	25.47 +- 0.184	26.34 +- 0.286
8	82.52 +- 0.000	48.64 +- 1.354	5.60 +- 0.378	15.72 +- 0.571	12.56 +- 1.182
16	46.16 +- 0.000	24.46 +- 1.149	5.85 +- 0.488	9.49 +- 0.526	6.35 +- 0.916
32	28.32 +- 0.000	12.30 +- 0.685	6.91 +- 0.781	5.50 +- 0.462	3.61 +- 0.646
64	21.43 +- 0.001	6.14 +- 0.493	9.38 +- 0.723	3.16 +- 0.355	2.74 +- 0.377

BASE+VC-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	505.87 +- 0.000	378.59 +- 0.000	0.38 +- 0.000	32.13 +- 0.000	94.78 +- 0.000
2	540.48 +- 0.000	202.84 +- 7.445	260.43 +- 7.349	20.99 +- 0.160	56.22 +- 0.256
4	421.37 +- 0.000	102.53 +- 25.873	276.68 +- 31.377	12.64 +- 1.468	29.51 +- 4.132
8	312.85 +- 0.000	51.14 +- 14.202	238.84 +- 16.233	7.68 +- 1.034	15.19 +- 2.191
16	230.36 +- 0.000	25.26 +- 14.803	192.95 +- 19.098	4.53 +- 1.538	7.63 +- 3.304
32	190.53 +- 0.000	12.79 +- 11.144	171.12 +- 15.742	2.56 +- 1.567	4.05 +- 3.361
64	156.97 +- 0.001	6.41 +- 9.030	147.24 +- 13.078	1.40 +- 1.256	1.93 +- 3.680

Table B.3: (continued)

BASE+VC-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	508.64 +- 0.000	380.57 +- 0.000	0.85 +- 0.000	32.13 +- 0.000	95.09 +- 0.000
2	285.24 +- 0.000	193.39 +- 6.172	21.19 +- 5.904	19.06 +- 0.078	51.60 +- 0.346
4	213.40 +- 0.000	98.16 +- 22.665	77.46 +- 27.179	10.68 +- 1.248	27.10 +- 3.343
8	135.36 +- 0.000	49.56 +- 15.172	65.92 +- 17.134	5.93 +- 0.734	13.96 +- 2.010
16	102.71 +- 0.000	24.87 +- 15.044	67.20 +- 18.860	3.41 +- 1.171	7.23 +- 3.123
32	73.15 +- 0.000	12.42 +- 10.816	54.96 +- 14.824	1.95 +- 1.244	3.82 +- 3.063
64	63.12 +- 0.001	6.17 +- 8.535	53.93 +- 11.649	0.92 +- 0.883	2.10 +- 2.439

BASE+VC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	530.00 +- 0.000	380.38 +- 0.000	0.40 +- 0.000	48.34 +- 0.000	100.88 +- 0.000
2	294.22 +- 0.000	191.94 +- 5.899	22.33 +- 5.565	28.00 +- 0.057	51.95 +- 0.391
4	204.45 +- 0.000	97.95 +- 22.726	62.86 +- 27.918	15.40 +- 1.692	28.24 +- 3.582
8	142.84 +- 0.000	49.37 +- 15.243	70.63 +- 17.469	8.42 +- 0.993	14.42 +- 2.057
16	105.28 +- 0.000	24.66 +- 15.236	68.68 +- 19.579	4.76 +- 1.602	7.18 +- 3.265
32	73.17 +- 0.000	12.31 +- 10.778	54.48 +- 15.268	2.67 +- 1.663	3.71 +- 3.157
64	67.35 +- 0.001	6.18 +- 8.555	57.87 +- 12.022	1.33 +- 1.234	1.97 +- 2.475

BASE+VC+WSHR-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	520.35 +- 0.000	384.91 +- 0.000	0.00 +- 0.000	32.70 +- 0.000	102.73 +- 0.000
2	450.49 +- 0.000	199.27 +- 0.576	159.70 +- 0.397	22.28 +- 0.026	69.24 +- 0.153
4	264.57 +- 0.000	99.69 +- 0.173	109.42 +- 0.430	15.62 +- 0.044	39.84 +- 0.227
8	143.90 +- 0.000	49.48 +- 0.190	62.62 +- 0.748	10.41 +- 0.078	21.39 +- 0.584
16	78.43 +- 0.000	24.63 +- 0.107	35.91 +- 0.578	6.77 +- 0.119	11.12 +- 0.512
32	43.41 +- 0.000	12.33 +- 0.084	21.05 +- 0.480	4.35 +- 0.133	5.69 +- 0.402
64	24.91 +- 0.001	6.18 +- 0.049	13.26 +- 0.281	2.50 +- 0.153	2.97 +- 0.240

BASE+VC+WSHR-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	523.69 +- 0.000	383.58 +- 0.000	0.00 +- 0.000	32.70 +- 0.000	107.41 +- 0.000
2	277.18 +- 0.000	192.37 +- 0.039	0.42 +- 0.108	19.50 +- 0.002	64.89 +- 0.067
4	147.15 +- 0.000	96.75 +- 0.082	1.13 +- 0.225	11.64 +- 0.034	37.62 +- 0.176
8	77.89 +- 0.000	48.49 +- 0.067	1.97 +- 0.538	7.03 +- 0.163	20.40 +- 0.490
16	42.16 +- 0.000	24.43 +- 0.077	2.74 +- 0.535	4.34 +- 0.066	10.66 +- 0.484
32	23.71 +- 0.000	12.22 +- 0.084	3.22 +- 0.443	2.69 +- 0.075	5.58 +- 0.361
64	14.72 +- 0.001	6.16 +- 0.066	4.08 +- 0.331	1.49 +- 0.109	2.98 +- 0.268

Table B.3: (continued)

BASE+VC+WSHR-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	529.96 +- 0.000	382.89 +- 0.000	0.00 +- 0.000	49.19 +- 0.000	97.88 +- 0.000
2	279.95 +- 0.000	192.01 +- 0.345	0.63 +- 0.149	28.75 +- 0.028	58.57 +- 0.168
4	148.52 +- 0.000	96.54 +- 0.167	1.15 +- 0.171	17.21 +- 0.063	33.61 +- 0.135
8	78.46 +- 0.000	48.40 +- 0.103	1.94 +- 0.451	10.35 +- 0.114	17.77 +- 0.392
16	42.26 +- 0.000	24.25 +- 0.086	2.48 +- 0.447	6.26 +- 0.103	9.27 +- 0.406
32	23.77 +- 0.000	12.18 +- 0.068	3.10 +- 0.401	3.82 +- 0.135	4.66 +- 0.317
64	14.72 +- 0.001	6.14 +- 0.065	4.05 +- 0.275	2.07 +- 0.133	2.46 +- 0.222

BASE+VC+WSHRLOC-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	522.75 +- 0.000	385.11 +- 0.000	0.00 +- 0.000	32.70 +- 0.000	104.94 +- 0.000
2	502.22 +- 0.000	200.57 +- 0.059	212.71 +- 0.561	22.20 +- 0.040	66.73 +- 0.462
4	278.74 +- 0.000	99.70 +- 0.228	124.36 +- 0.480	15.38 +- 0.032	39.30 +- 0.315
8	150.18 +- 0.000	49.59 +- 0.354	69.92 +- 0.725	9.76 +- 0.109	20.91 +- 0.596
16	81.50 +- 0.000	24.72 +- 0.337	39.75 +- 0.880	6.26 +- 0.120	10.78 +- 0.539
32	44.93 +- 0.000	12.37 +- 0.226	23.37 +- 0.539	3.75 +- 0.120	5.44 +- 0.402
64	26.19 +- 0.001	6.19 +- 0.310	15.13 +- 0.478	2.12 +- 0.167	2.75 +- 0.270

BASE+VC+WSHRLOC-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	524.13 +- 0.000	383.69 +- 0.000	0.00 +- 0.000	32.70 +- 0.000	107.74 +- 0.000
2	273.00 +- 0.000	191.58 +- 0.115	1.29 +- 0.634	19.47 +- 0.032	60.66 +- 0.488
4	145.20 +- 0.000	96.36 +- 0.213	1.56 +- 0.329	11.60 +- 0.042	35.68 +- 0.180
8	76.77 +- 0.000	48.33 +- 0.121	2.27 +- 0.349	6.81 +- 0.140	19.36 +- 0.364
16	41.81 +- 0.000	24.36 +- 0.246	3.29 +- 0.559	4.11 +- 0.082	10.05 +- 0.439
32	23.37 +- 0.000	12.20 +- 0.130	3.49 +- 0.433	2.49 +- 0.113	5.19 +- 0.328
64	14.60 +- 0.001	6.14 +- 0.222	4.45 +- 0.450	1.32 +- 0.117	2.70 +- 0.257

BASE+VC+WSHRLOC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	529.73 +- 0.000	382.66 +- 0.000	0.00 +- 0.000	49.19 +- 0.000	97.88 +- 0.000
2	276.95 +- 0.000	191.21 +- 0.006	1.27 +- 0.463	28.71 +- 0.052	55.76 +- 0.416
4	147.22 +- 0.000	96.18 +- 0.165	1.32 +- 0.352	17.14 +- 0.049	32.58 +- 0.209
8	77.99 +- 0.000	48.31 +- 0.328	2.28 +- 0.373	10.04 +- 0.171	17.37 +- 0.325
16	41.86 +- 0.000	24.20 +- 0.105	2.72 +- 0.373	5.98 +- 0.112	8.96 +- 0.408
32	23.51 +- 0.000	12.15 +- 0.160	3.47 +- 0.425	3.45 +- 0.143	4.43 +- 0.311
64	14.60 +- 0.001	6.11 +- 0.198	4.34 +- 0.299	1.88 +- 0.150	2.27 +- 0.230

BASE+VC+WSTL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	532.46 +- 0.000	384.27 +- 0.000	23.97 +- 0.000	32.46 +- 0.000	91.75 +- 0.000

Table B.3: (continued)

2	324.60 +- 0.000	193.68 +- 0.945	42.90 +- 1.152	21.97 +- 0.031	66.05 +- 0.176
4	201.57 +- 0.000	97.41 +- 0.512	52.78 +- 0.214	14.93 +- 0.103	36.46 +- 0.202
8	117.97 +- 0.000	48.96 +- 3.302	40.04 +- 2.464	9.43 +- 0.103	19.54 +- 0.800
16	66.30 +- 0.000	24.50 +- 3.082	25.68 +- 2.174	5.86 +- 0.215	10.26 +- 0.740
32	38.61 +- 0.000	12.29 +- 1.624	17.27 +- 1.117	3.62 +- 0.226	5.43 +- 0.538
64	23.54 +- 0.002	6.15 +- 0.587	12.31 +- 0.483	2.05 +- 0.170	3.04 +- 0.328

BASE+VC+WSTL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	532.37 +- 0.000	381.73 +- 0.000	24.33 +- 0.000	32.46 +- 0.000	93.84 +- 0.000
2	276.96 +- 0.000	191.54 +- 0.227	1.95 +- 0.093	19.33 +- 0.022	64.14 +- 0.298
4	144.11 +- 0.000	96.02 +- 0.196	1.59 +- 0.137	11.38 +- 0.094	35.12 +- 0.239
8	75.23 +- 0.000	47.85 +- 0.774	2.23 +- 0.357	6.55 +- 0.188	18.60 +- 0.867
16	40.46 +- 0.000	24.04 +- 0.731	2.91 +- 0.696	3.93 +- 0.246	9.59 +- 0.867
32	22.92 +- 0.000	12.11 +- 0.507	3.47 +- 0.785	2.31 +- 0.289	5.03 +- 0.670
64	14.19 +- 0.002	6.08 +- 0.541	4.12 +- 0.736	1.22 +- 0.191	2.77 +- 0.489

BASE+VC+WSTL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	540.85 +- 0.000	382.34 +- 0.000	24.03 +- 0.000	48.83 +- 0.000	85.65 +- 0.000
2	279.57 +- 0.000	191.39 +- 0.170	11.47 +- 0.468	28.46 +- 0.009	48.25 +- 0.648
4	146.63 +- 0.000	95.92 +- 0.313	6.78 +- 0.155	16.87 +- 0.095	27.06 +- 0.245
8	77.84 +- 0.000	48.16 +- 1.154	6.09 +- 0.269	9.91 +- 0.309	13.68 +- 1.009
16	42.98 +- 0.000	24.15 +- 0.932	5.76 +- 0.380	6.03 +- 0.255	7.05 +- 0.844
32	26.43 +- 0.000	12.20 +- 0.639	6.75 +- 0.658	3.54 +- 0.238	3.95 +- 0.558
64	19.73 +- 0.003	6.10 +- 0.538	8.86 +- 0.632	1.95 +- 0.157	2.82 +- 0.373

BASE+VC+DATA-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	494.97 +- 0.000	380.20 +- 0.000	0.00 +- 0.000	32.14 +- 0.000	82.62 +- 0.000
2	488.62 +- 0.000	200.15 +- 0.313	213.44 +- 0.724	21.80 +- 0.065	53.24 +- 0.347
4	357.77 +- 0.000	99.49 +- 0.677	212.65 +- 1.055	14.70 +- 0.116	30.93 +- 0.317
8	202.00 +- 0.000	49.69 +- 1.091	126.68 +- 2.542	9.04 +- 0.297	16.59 +- 1.206
16	124.00 +- 0.000	24.67 +- 1.613	85.09 +- 3.154	5.51 +- 0.356	8.73 +- 1.247
32	72.34 +- 0.000	12.36 +- 1.228	52.29 +- 2.525	3.28 +- 0.366	4.41 +- 1.013
64	46.62 +- 0.001	6.22 +- 0.891	36.16 +- 1.832	1.88 +- 0.287	2.36 +- 0.803

BASE+VC+DATA-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	494.18 +- 0.000	379.90 +- 0.000	0.00 +- 0.000	32.14 +- 0.000	82.13 +- 0.000
2	264.14 +- 0.000	191.33 +- 0.531	5.84 +- 0.799	19.23 +- 0.058	47.74 +- 0.210
4	143.62 +- 0.000	96.12 +- 0.451	8.96 +- 0.737	11.24 +- 0.095	27.30 +- 0.224
8	79.51 +- 0.000	48.16 +- 1.724	10.22 +- 3.104	6.44 +- 0.315	14.69 +- 1.115

Table B.3: (continued)

16	44.61 +- 0.000	24.12 +- 0.958	8.92 +- 2.210	3.73 +- 0.298	7.84 +- 1.074
32	28.04 +- 0.000	12.15 +- 1.078	9.67 +- 2.172	2.14 +- 0.297	4.08 +- 0.861
64	19.63 +- 0.001	6.13 +- 0.764	10.09 +- 1.642	1.16 +- 0.241	2.25 +- 0.698

BASE+VC+DATA-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	514.17 +- 0.000	378.43 +- 0.000	0.00 +- 0.000	48.36 +- 0.000	87.39 +- 0.000
2	274.29 +- 0.000	190.41 +- 0.629	5.43 +- 0.740	28.36 +- 0.093	50.10 +- 0.019
4	149.73 +- 0.000	95.70 +- 0.427	9.64 +- 0.801	16.58 +- 0.166	27.81 +- 0.249
8	82.59 +- 0.000	48.10 +- 1.529	10.27 +- 2.718	9.46 +- 0.432	14.75 +- 1.037
16	46.28 +- 0.000	24.00 +- 0.980	9.34 +- 2.220	5.39 +- 0.392	7.54 +- 0.966
32	28.04 +- 0.000	12.06 +- 0.864	9.20 +- 1.813	2.98 +- 0.331	3.81 +- 0.730
64	21.31 +- 0.001	6.12 +- 0.755	11.43 +- 1.691	1.66 +- 0.312	2.10 +- 0.721

BASE+NOFS+WSHR-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	520.59 +- 0.000	385.12 +- 0.000	0.00 +- 0.000	32.70 +- 0.000	102.77 +- 0.000
2	452.39 +- 0.000	197.74 +- 0.031	162.96 +- 0.258	23.76 +- 0.013	67.93 +- 0.215
4	265.69 +- 0.000	99.02 +- 0.186	110.48 +- 0.369	17.60 +- 0.025	38.59 +- 0.219
8	148.67 +- 0.000	49.34 +- 0.181	66.84 +- 0.676	12.07 +- 0.143	20.43 +- 0.485
16	82.12 +- 0.000	24.63 +- 0.131	39.07 +- 0.585	8.02 +- 0.098	10.40 +- 0.512
32	46.42 +- 0.000	12.32 +- 0.058	23.80 +- 0.450	5.08 +- 0.173	5.22 +- 0.327
64	26.40 +- 0.001	6.19 +- 0.055	14.50 +- 0.331	2.98 +- 0.208	2.74 +- 0.268

BASE+NOFS+WSHR-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	523.69 +- 0.000	383.56 +- 0.000	0.00 +- 0.000	32.70 +- 0.000	107.43 +- 0.000
2	459.09 +- 0.000	199.85 +- 0.131	165.54 +- 0.287	20.69 +- 0.028	73.01 +- 0.128
4	266.96 +- 0.000	99.95 +- 0.317	112.14 +- 0.353	13.10 +- 0.042	41.77 +- 0.208
8	144.06 +- 0.000	49.78 +- 0.134	63.52 +- 0.837	8.31 +- 0.115	22.45 +- 0.691
16	77.91 +- 0.000	24.92 +- 0.143	36.21 +- 0.733	5.25 +- 0.089	11.53 +- 0.599
32	42.75 +- 0.000	12.41 +- 0.074	21.18 +- 0.504	3.34 +- 0.112	5.83 +- 0.451
64	24.19 +- 0.001	6.23 +- 0.065	12.99 +- 0.406	1.88 +- 0.109	3.09 +- 0.327

BASE+NOFS+WSHR-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	529.98 +- 0.000	382.81 +- 0.000	0.00 +- 0.000	49.19 +- 0.000	97.98 +- 0.000
2	283.07 +- 0.000	192.85 +- 0.194	0.64 +- 0.322	29.91 +- 0.038	59.67 +- 0.090
4	151.42 +- 0.000	96.95 +- 0.128	1.00 +- 0.167	18.58 +- 0.057	34.89 +- 0.107
8	80.90 +- 0.000	48.69 +- 0.088	2.02 +- 0.614	11.42 +- 0.252	18.78 +- 0.445
16	44.26 +- 0.000	24.40 +- 0.095	2.88 +- 0.530	7.06 +- 0.140	9.92 +- 0.428
32	25.31 +- 0.000	12.26 +- 0.078	3.61 +- 0.410	4.38 +- 0.129	5.04 +- 0.308
64	16.11 +- 0.001	6.16 +- 0.066	4.73 +- 0.344	2.47 +- 0.135	2.75 +- 0.279

Table B.4: Radiosity execution times (in seconds) on the SGI Origin 2000.

BASE-CC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	64.85 +- 0.000	61.74 +- 0.000	0.00 +- 0.000	0.00 +- 0.000	3.11 +- 0.000
2	36.72 +- 0.269	34.85 +- 0.260	0.32 +- 0.000	0.00 +- 0.000	1.56 +- 0.009
4	19.25 +- 0.198	18.28 +- 0.240	0.20 +- 0.000	0.00 +- 0.000	0.78 +- 0.047
8	12.55 +- 0.085	12.09 +- 0.102	0.03 +- 0.000	0.00 +- 0.000	0.42 +- 0.027
16	12.18 +- 2.480	11.87 +- 2.428	0.10 +- 0.000	0.00 +- 0.000	0.21 +- 0.062

BASE-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	66.83 +- 0.000	58.55 +- 0.000	0.00 +- 0.000	6.05 +- 0.000	2.23 +- 0.000
2	70.58 +- 0.004	25.57 +- 0.271	34.80 +- 0.309	6.90 +- 0.054	3.32 +- 0.012
4	61.49 +- 0.049	12.59 +- 2.893	41.14 +- 3.667	5.21 +- 0.521	2.55 +- 0.236
8	50.04 +- 0.296	8.24 +- 2.245	36.78 +- 2.587	3.39 +- 0.304	1.63 +- 0.156
16	43.62 +- 0.273	10.75 +- 2.003	29.76 +- 2.842	2.12 +- 0.610	0.99 +- 0.327

BASE-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	67.77 +- 0.000	59.49 +- 0.000	0.00 +- 0.000	6.05 +- 0.000	2.23 +- 0.000
2	69.11 +- 0.029	27.48 +- 0.805	34.50 +- 0.873	4.19 +- 0.047	2.94 +- 0.008
4	62.76 +- 1.641	14.67 +- 3.647	43.34 +- 4.766	2.57 +- 0.297	2.18 +- 0.214
8	47.29 +- 0.801	7.51 +- 2.457	36.90 +- 3.059	1.51 +- 0.199	1.37 +- 0.136
16	35.71 +- 0.236	4.06 +- 2.244	29.97 +- 2.852	0.87 +- 0.299	0.82 +- 0.279

BASE-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	72.30 +- 0.000	61.99 +- 0.000	0.00 +- 0.000	8.08 +- 0.000	2.23 +- 0.000
2	41.50 +- 0.306	28.86 +- 1.058	3.31 +- 1.409	7.48 +- 0.058	1.85 +- 0.014
4	25.08 +- 0.274	13.40 +- 3.014	5.77 +- 3.742	4.83 +- 0.526	1.09 +- 0.121
8	17.76 +- 0.333	7.13 +- 2.371	7.13 +- 2.832	2.92 +- 0.285	0.59 +- 0.082
16	13.37 +- 0.291	3.61 +- 1.880	7.72 +- 2.663	1.74 +- 0.544	0.30 +- 0.143

BASE+WSHR-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	68.05 +- 0.000	58.75 +- 0.000	0.00 +- 0.000	6.16 +- 0.000	3.14 +- 0.000
2	66.98 +- 0.113	27.03 +- 0.131	22.91 +- 0.252	9.31 +- 0.004	7.72 +- 0.004
4	47.74 +- 0.159	18.62 +- 0.731	15.31 +- 0.811	7.72 +- 0.017	6.09 +- 0.017
8	32.18 +- 0.132	14.09 +- 0.543	8.72 +- 0.555	5.42 +- 0.106	3.95 +- 0.104
16	29.64 +- 0.155	15.05 +- 0.562	8.67 +- 0.524	3.55 +- 0.102	2.38 +- 0.091

Table B.4: (continued)

BASE+WSHR-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	68.01 +- 0.000	58.70 +- 0.000	0.00 +- 0.000	6.16 +- 0.000	3.14 +- 0.000
2	64.97 +- 0.058	30.13 +- 0.216	23.00 +- 0.263	4.76 +- 0.008	7.09 +- 0.004
4	44.42 +- 0.090	19.91 +- 0.207	15.99 +- 0.170	3.08 +- 0.011	5.45 +- 0.015
8	28.57 +- 0.058	13.84 +- 0.464	9.37 +- 0.519	1.90 +- 0.020	3.45 +- 0.067
16	24.15 +- 0.180	12.67 +- 0.277	8.26 +- 0.300	1.18 +- 0.026	2.04 +- 0.085

BASE+WSHR-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	78.79 +- 0.000	67.43 +- 0.000	0.00 +- 0.000	8.22 +- 0.000	3.14 +- 0.000
2	40.61 +- 0.030	25.94 +- 0.052	0.16 +- 0.057	9.27 +- 0.021	5.24 +- 0.004
4	24.98 +- 0.076	14.12 +- 0.370	0.54 +- 0.307	6.79 +- 0.009	3.53 +- 0.014
8	16.33 +- 0.116	8.85 +- 0.428	1.00 +- 0.436	4.49 +- 0.053	2.00 +- 0.076
16	10.77 +- 0.219	5.00 +- 0.138	1.91 +- 0.341	2.80 +- 0.045	1.06 +- 0.078

BASE+WSHRLOC-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	68.04 +- 0.000	58.74 +- 0.000	0.00 +- 0.000	6.16 +- 0.000	3.14 +- 0.000
2	68.47 +- 0.002	25.23 +- 0.097	26.20 +- 0.099	9.32 +- 0.000	7.72 +- 0.004
4	46.27 +- 0.105	16.60 +- 0.193	15.65 +- 0.206	7.85 +- 0.036	6.17 +- 0.020
8	30.73 +- 0.171	11.40 +- 0.441	9.93 +- 0.526	5.44 +- 0.071	3.96 +- 0.085
16	24.76 +- 0.160	9.16 +- 0.480	9.85 +- 0.509	3.43 +- 0.046	2.32 +- 0.077

BASE+WSHRLOC-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	69.28 +- 0.000	59.98 +- 0.000	0.00 +- 0.000	6.16 +- 0.000	3.14 +- 0.000
2	67.33 +- 0.025	28.61 +- 0.105	26.83 +- 0.091	4.77 +- 0.006	7.11 +- 0.005
4	46.00 +- 0.142	20.62 +- 0.374	16.80 +- 0.306	3.09 +- 0.013	5.49 +- 0.018
8	28.46 +- 0.087	13.71 +- 0.523	9.49 +- 0.576	1.87 +- 0.034	3.39 +- 0.074
16	23.09 +- 0.138	11.06 +- 0.411	8.87 +- 0.410	1.16 +- 0.019	2.00 +- 0.084

BASE+WSHRLOC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	70.67 +- 0.000	59.31 +- 0.000	0.00 +- 0.000	8.22 +- 0.000	3.14 +- 0.000
2	40.05 +- 0.028	25.20 +- 0.049	0.29 +- 0.094	9.32 +- 0.014	5.25 +- 0.004
4	24.15 +- 0.115	13.21 +- 0.041	0.46 +- 0.175	6.94 +- 0.033	3.55 +- 0.015
8	15.57 +- 0.170	8.04 +- 0.365	1.19 +- 0.487	4.36 +- 0.041	1.97 +- 0.063
16	9.91 +- 0.174	4.49 +- 0.076	1.64 +- 0.218	2.73 +- 0.033	1.05 +- 0.082

Table B.4: (continued)

BASE+WSTL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	68.08 +- 0.000	58.73 +- 0.000	0.10 +- 0.000	6.12 +- 0.000	3.13 +- 0.000
2	56.55 +- 0.008	26.85 +- 0.016	8.06 +- 0.020	8.37 +- 0.008	13.27 +- 0.004
4	37.29 +- 0.055	17.90 +- 0.162	7.25 +- 0.138	6.57 +- 0.032	5.57 +- 0.023
8	24.40 +- 0.071	11.81 +- 1.055	5.05 +- 0.740	4.14 +- 0.172	3.40 +- 0.150
16	25.60 +- 0.202	13.45 +- 0.637	6.68 +- 0.316	2.93 +- 0.105	2.54 +- 0.126

BASE+WSTL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	67.96 +- 0.000	58.61 +- 0.000	0.10 +- 0.000	6.12 +- 0.000	3.13 +- 0.000
2	51.60 +- 0.016	29.56 +- 0.120	8.46 +- 0.142	4.52 +- 0.002	9.07 +- 0.004
4	32.60 +- 0.051	17.75 +- 0.159	7.33 +- 0.145	2.84 +- 0.015	4.68 +- 0.030
8	23.36 +- 0.120	12.89 +- 0.857	5.54 +- 0.722	1.72 +- 0.038	3.21 +- 0.121
16	23.14 +- 0.208	13.63 +- 0.669	6.27 +- 0.382	1.05 +- 0.033	2.19 +- 0.132

BASE+WSTL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	72.93 +- 0.000	61.44 +- 0.000	0.00 +- 0.000	8.16 +- 0.000	3.33 +- 0.000
2	39.89 +- 0.026	26.83 +- 0.097	0.30 +- 0.107	8.56 +- 0.037	4.20 +- 0.001
4	22.57 +- 0.086	13.49 +- 0.044	0.13 +- 0.015	6.20 +- 0.041	2.75 +- 0.018
8	14.19 +- 0.138	8.57 +- 0.312	0.05 +- 0.012	3.96 +- 0.128	1.60 +- 0.069
16	8.95 +- 0.322	5.39 +- 0.429	0.08 +- 0.016	2.49 +- 0.091	0.99 +- 0.076

BASE+VC-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	68.34 +- 0.000	60.05 +- 0.000	0.00 +- 0.000	6.05 +- 0.000	2.23 +- 0.000
2	68.02 +- 0.084	27.73 +- 0.539	33.85 +- 0.638	4.27 +- 0.026	2.16 +- 0.012
4	50.64 +- 0.099	14.38 +- 3.109	32.17 +- 3.606	2.69 +- 0.293	1.40 +- 0.148
8	39.36 +- 0.531	7.62 +- 1.985	29.26 +- 2.252	1.68 +- 0.221	0.81 +- 0.098
16	29.85 +- 0.321	5.91 +- 2.045	22.47 +- 2.689	1.01 +- 0.337	0.46 +- 0.183

BASE+VC-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	66.69 +- 0.000	58.41 +- 0.000	0.00 +- 0.000	6.05 +- 0.000	2.23 +- 0.000
2	36.69 +- 0.008	28.72 +- 0.272	2.52 +- 0.302	3.61 +- 0.035	1.84 +- 0.012
4	23.70 +- 0.192	14.52 +- 3.137	6.08 +- 3.644	2.02 +- 0.237	1.08 +- 0.120
8	14.56 +- 0.135	7.16 +- 2.078	5.69 +- 2.244	1.13 +- 0.139	0.58 +- 0.081
16	12.99 +- 0.609	3.82 +- 2.081	8.23 +- 2.797	0.64 +- 0.219	0.30 +- 0.143

BASE+VC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	69.72 +- 0.000	59.40 +- 0.000	0.00 +- 0.000	8.08 +- 0.000	2.23 +- 0.000

Table B.4: (continued)

2	41.22 +- 0.053	31.61 +- 0.261	2.97 +- 0.322	4.80 +- 0.022	1.84 +- 0.013
4	25.23 +- 0.115	15.19 +- 3.129	6.29 +- 3.591	2.66 +- 0.292	1.08 +- 0.120
8	16.01 +- 0.148	7.90 +- 2.021	6.04 +- 2.179	1.48 +- 0.172	0.58 +- 0.081
16	13.22 +- 0.592	3.60 +- 2.186	8.48 +- 2.946	0.85 +- 0.279	0.30 +- 0.143

BASE+VC+WSHR-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	68.57 +- 0.000	59.27 +- 0.000	0.00 +- 0.000	6.16 +- 0.000	3.14 +- 0.000
2	60.60 +- 0.028	29.44 +- 0.014	20.80 +- 0.035	4.66 +- 0.003	5.70 +- 0.004
4	40.98 +- 1.006	18.90 +- 1.115	14.28 +- 2.118	3.60 +- 0.004	4.20 +- 0.015
8	26.42 +- 0.484	12.54 +- 0.452	8.78 +- 0.902	2.53 +- 0.033	2.58 +- 0.046
16	20.61 +- 0.149	8.79 +- 0.239	8.65 +- 0.313	1.70 +- 0.031	1.48 +- 0.082

BASE+VC+WSHR-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	68.83 +- 0.000	59.53 +- 0.000	0.00 +- 0.000	6.16 +- 0.000	3.14 +- 0.000
2	37.60 +- 0.062	28.56 +- 0.024	0.14 +- 0.048	3.68 +- 0.007	5.22 +- 0.004
4	22.98 +- 0.275	16.35 +- 0.297	0.93 +- 0.574	2.19 +- 0.012	3.51 +- 0.015
8	13.68 +- 0.144	9.79 +- 0.126	0.61 +- 0.242	1.30 +- 0.029	1.98 +- 0.066
16	9.24 +- 0.249	5.84 +- 0.121	1.54 +- 0.314	0.82 +- 0.013	1.05 +- 0.081

BASE+VC+WSHR-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	70.98 +- 0.000	59.62 +- 0.000	0.00 +- 0.000	8.22 +- 0.000	3.14 +- 0.000
2	39.32 +- 0.004	29.06 +- 0.071	0.11 +- 0.057	4.93 +- 0.007	5.22 +- 0.003
4	23.62 +- 0.294	16.41 +- 0.217	0.67 +- 0.447	3.02 +- 0.015	3.51 +- 0.015
8	13.84 +- 0.074	9.57 +- 0.053	0.45 +- 0.138	1.84 +- 0.031	1.98 +- 0.069
16	9.71 +- 0.229	5.88 +- 0.136	1.62 +- 0.344	1.15 +- 0.022	1.05 +- 0.086

BASE+VC+WSHRLOC-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	69.33 +- 0.000	60.03 +- 0.000	0.00 +- 0.000	6.16 +- 0.000	3.14 +- 0.000
2	66.16 +- 0.035	27.38 +- 0.074	28.44 +- 0.121	4.64 +- 0.010	5.69 +- 0.004
4	43.11 +- 0.239	18.21 +- 0.315	17.19 +- 0.529	3.53 +- 0.008	4.18 +- 0.014
8	26.37 +- 0.298	10.72 +- 0.313	10.78 +- 0.578	2.36 +- 0.026	2.50 +- 0.072
16	19.78 +- 0.161	6.75 +- 0.287	10.11 +- 0.351	1.52 +- 0.021	1.40 +- 0.081

BASE+VC+WSHRLOC-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	68.34 +- 0.000	59.04 +- 0.000	0.00 +- 0.000	6.16 +- 0.000	3.14 +- 0.000
2	36.80 +- 0.009	27.77 +- 0.007	0.15 +- 0.009	3.67 +- 0.003	5.22 +- 0.004
4	22.18 +- 0.087	16.20 +- 0.085	0.30 +- 0.140	2.17 +- 0.007	3.52 +- 0.014
8	13.61 +- 0.136	9.75 +- 0.141	0.59 +- 0.213	1.28 +- 0.025	1.99 +- 0.084

Table B.4: (continued)

16	9.10 +- 0.322	5.64 +- 0.141	1.60 +- 0.392	0.80 +- 0.015	1.05 +- 0.070

BASE+VC+WSHRLOC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	71.10 +- 0.000	59.74 +- 0.000	0.00 +- 0.000	8.22 +- 0.000	3.14 +- 0.000
2	38.86 +- 0.013	28.56 +- 0.027	0.17 +- 0.037	4.91 +- 0.001	5.22 +- 0.004
4	26.04 +- 0.319	18.23 +- 0.629	1.30 +- 0.960	2.99 +- 0.016	3.52 +- 0.015
8	15.78 +- 0.267	10.48 +- 0.373	1.58 +- 0.644	1.75 +- 0.023	1.97 +- 0.076
16	9.87 +- 0.267	6.33 +- 0.154	1.40 +- 0.382	1.09 +- 0.022	1.05 +- 0.082

BASE+VC+WSTL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	71.30 +- 0.000	61.95 +- 0.000	0.10 +- 0.000	6.12 +- 0.000	3.13 +- 0.000
2	49.87 +- 0.057	33.89 +- 0.465	5.43 +- 0.531	4.58 +- 0.005	5.96 +- 0.005
4	31.49 +- 0.696	17.75 +- 0.447	6.32 +- 1.057	3.39 +- 0.061	4.02 +- 0.038
8	21.09 +- 0.423	11.15 +- 0.230	5.19 +- 0.465	2.24 +- 0.044	2.51 +- 0.051
16	18.39 +- 0.186	8.96 +- 0.680	6.36 +- 0.531	1.43 +- 0.057	1.64 +- 0.100

BASE+VC+WSTL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	74.62 +- 0.000	65.27 +- 0.000	0.10 +- 0.000	6.12 +- 0.000	3.13 +- 0.000
2	41.33 +- 0.053	31.39 +- 0.312	0.42 +- 0.329	3.65 +- 0.032	5.86 +- 0.003
4	22.24 +- 0.149	16.94 +- 0.128	0.25 +- 0.108	2.14 +- 0.018	2.91 +- 0.012
8	13.27 +- 0.254	10.10 +- 0.186	0.25 +- 0.132	1.25 +- 0.042	1.67 +- 0.061
16	8.44 +- 0.276	5.87 +- 0.166	0.89 +- 0.219	0.75 +- 0.038	0.94 +- 0.080

BASE+VC+WSTL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	79.37 +- 0.000	67.88 +- 0.000	0.00 +- 0.000	8.16 +- 0.000	3.33 +- 0.000
2	38.54 +- 0.017	29.31 +- 0.093	0.18 +- 0.077	4.87 +- 0.003	4.18 +- 0.001
4	23.57 +- 0.099	17.85 +- 0.102	0.05 +- 0.004	2.94 +- 0.018	2.73 +- 0.015
8	12.96 +- 0.121	9.60 +- 0.177	0.04 +- 0.017	1.74 +- 0.070	1.58 +- 0.088
16	7.47 +- 0.106	5.41 +- 0.147	0.10 +- 0.026	1.06 +- 0.059	0.90 +- 0.080

BASE+VC+DATA-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	66.13 +- 0.000	57.81 +- 0.000	0.00 +- 0.000	6.06 +- 0.000	2.26 +- 0.000
2	68.59 +- 0.037	28.57 +- 0.286	31.36 +- 0.258	4.54 +- 0.006	4.11 +- 0.003
4	46.84 +- 0.905	16.85 +- 1.175	23.62 +- 2.071	3.32 +- 0.019	3.04 +- 0.016
8	27.99 +- 0.146	10.28 +- 0.468	13.76 +- 0.676	2.13 +- 0.061	1.82 +- 0.079
16	22.01 +- 0.173	7.20 +- 0.401	12.48 +- 0.581	1.31 +- 0.073	1.02 +- 0.091

Table B.4: (continued)

BASE+VC+DATA-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	66.40 +- 0.000	58.08 +- 0.000	0.00 +- 0.000	6.06 +- 0.000	2.26 +- 0.000
2	36.51 +- 0.014	28.40 +- 0.305	0.81 +- 0.308	3.63 +- 0.014	3.67 +- 0.003
4	21.60 +- 0.362	15.74 +- 0.506	1.30 +- 0.851	2.11 +- 0.021	2.45 +- 0.014
8	12.27 +- 0.107	8.55 +- 0.278	1.13 +- 0.447	1.21 +- 0.051	1.38 +- 0.088
16	8.09 +- 0.180	5.14 +- 0.164	1.50 +- 0.194	0.71 +- 0.054	0.73 +- 0.084

BASE+VC+DATA-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	69.63 +- 0.000	59.29 +- 0.000	0.00 +- 0.000	8.08 +- 0.000	2.26 +- 0.000
2	38.36 +- 0.017	29.03 +- 0.272	0.80 +- 0.314	4.86 +- 0.021	3.67 +- 0.003
4	22.04 +- 0.076	15.59 +- 0.445	1.10 +- 0.419	2.90 +- 0.024	2.45 +- 0.014
8	13.17 +- 0.104	8.96 +- 0.340	1.16 +- 0.460	1.67 +- 0.068	1.38 +- 0.080
16	8.87 +- 0.192	5.31 +- 0.215	1.86 +- 0.282	0.97 +- 0.072	0.73 +- 0.077

BASE+NOFS+WSHR-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	68.35 +- 0.000	59.04 +- 0.000	0.00 +- 0.000	6.16 +- 0.000	3.14 +- 0.000
2	61.13 +- 0.033	28.72 +- 0.013	21.17 +- 0.052	5.20 +- 0.002	6.04 +- 0.004
4	39.87 +- 0.244	17.65 +- 0.383	13.22 +- 0.618	4.32 +- 0.002	4.68 +- 0.014
8	26.89 +- 0.217	12.55 +- 0.245	8.25 +- 0.491	3.13 +- 0.041	2.97 +- 0.107
16	21.36 +- 0.142	9.05 +- 0.324	8.42 +- 0.402	2.13 +- 0.032	1.75 +- 0.085

BASE+NOFS+WSHR-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	68.38 +- 0.000	59.08 +- 0.000	0.00 +- 0.000	6.16 +- 0.000	3.14 +- 0.000
2	60.47 +- 0.049	28.96 +- 0.047	21.93 +- 0.107	3.78 +- 0.006	5.80 +- 0.004
4	39.06 +- 0.096	18.42 +- 0.176	14.02 +- 0.148	2.29 +- 0.012	4.32 +- 0.015
8	25.69 +- 0.151	12.95 +- 0.311	8.65 +- 0.286	1.41 +- 0.023	2.68 +- 0.060
16	20.29 +- 0.182	9.33 +- 0.271	8.53 +- 0.321	0.88 +- 0.023	1.55 +- 0.083

BASE+NOFS+WSHR-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	71.67 +- 0.000	60.31 +- 0.000	0.00 +- 0.000	8.22 +- 0.000	3.14 +- 0.000
2	42.47 +- 0.040	30.69 +- 0.934	0.98 +- 0.970	5.56 +- 0.000	5.24 +- 0.004
4	23.75 +- 0.061	16.10 +- 0.154	0.26 +- 0.093	3.86 +- 0.012	3.53 +- 0.014
8	15.61 +- 0.191	10.13 +- 0.233	0.94 +- 0.399	2.54 +- 0.038	2.00 +- 0.068
16	11.67 +- 0.386	6.10 +- 0.186	2.86 +- 0.455	1.66 +- 0.022	1.06 +- 0.079

B.3 Gröbner

Tables B.5 and B.6 list the execution times for the Gröbner application on the Cray T3D and the SGI Origin 2000, respectively.

Measurements are presented for different numbers of processors, across the range of communication architectures, and for different levels of support of data locality and load balancing. The prefix of the configuration label denotes support for data locality and load balance: BASE, BASE+WSHR, BASE+WSTL, BASE+WSTLGBL, BASE+VC, BASE+VC+WSHR, BASE+VC+WSTL, and BASE+VC+WSTLGBL correspond to the baseline version of the application, and its augmentation with various load balancing policies and custom view coherence protocols described in Section 7.4. The TOTAL, COMPUTE, IDLE, COHERENCE, and OVERHEAD columns show the total execution time, and its breakdown into compute time, idle time, coherence time, and runtime overhead components. The data in the tables are displayed graphically in Figures 7.16–7.19.

Table B.5: Gröbner execution times (in seconds) on the Cray T3D.

BASE-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	517.82 +- 0.000	487.00 +- 0.000	7.18 +- 0.000	8.57 +- 0.000	15.06 +- 0.000
2	318.26 +- 0.000	284.87 +- 3.473	19.10 +- 2.916	5.33 +- 0.214	8.96 +- 0.343
4	191.12 +- 0.006	137.69 +- 10.381	43.55 +- 12.175	2.73 +- 0.123	7.14 +- 4.363
8	129.54 +- 0.005	66.94 +- 4.759	56.70 +- 7.244	1.55 +- 0.039	4.34 +- 4.582
16	97.72 +- 0.009	29.61 +- 10.057	64.86 +- 10.140	0.83 +- 0.107	2.42 +- 3.753
32	60.95 +- 0.052	10.51 +- 3.908	48.87 +- 4.319	0.41 +- 0.102	1.16 +- 2.212

BASE-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	535.24 +- 0.000	504.70 +- 0.000	6.58 +- 0.000	8.57 +- 0.000	15.40 +- 0.000
2	356.72 +- 0.000	316.32 +- 12.178	26.42 +- 12.809	5.28 +- 0.229	8.71 +- 0.402
4	189.08 +- 0.001	145.49 +- 6.054	34.07 +- 6.020	3.07 +- 0.105	6.45 +- 2.237
8	178.09 +- 0.006	99.39 +- 8.323	72.56 +- 10.268	1.71 +- 0.070	4.43 +- 4.135
16	248.95 +- 0.010	67.98 +- 21.826	176.42 +- 21.851	1.00 +- 0.114	3.54 +- 7.633
32	106.46 +- 0.068	19.21 +- 5.232	85.63 +- 5.899	0.44 +- 0.068	1.18 +- 3.665

BASE-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	515.89 +- 0.000	472.81 +- 0.000	8.58 +- 0.000	12.38 +- 0.000	22.12 +- 0.000
2	304.27 +- 0.000	253.58 +- 12.934	24.33 +- 6.691	8.96 +- 1.916	17.39 +- 4.327
4	193.95 +- 0.003	122.98 +- 10.289	50.16 +- 4.807	8.61 +- 1.070	12.19 +- 8.721
8	130.72 +- 0.011	55.24 +- 5.236	51.90 +- 17.036	5.22 +- 1.358	18.35 +- 14.142

Table B.5: (continued)

16	409.39 +- 0.062	82.90 +- 9.920	206.63 +- 76.788	6.86 +- 3.136	113.00 +- 74.432
32	106.46 +- 0.068	19.21 +- 5.232	85.63 +- 5.899	0.44 +- 0.068	1.18 +- 3.664

BASE+WSHR-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	517.82 +- 0.000	487.00 +- 0.000	7.18 +- 0.000	8.57 +- 0.000	15.06 +- 0.000
2	318.26 +- 0.000	284.87 +- 3.473	19.10 +- 2.916	5.33 +- 0.214	8.96 +- 0.343
4	191.12 +- 0.006	137.69 +- 10.381	43.55 +- 12.175	2.73 +- 0.123	7.14 +- 4.363
8	129.54 +- 0.005	66.94 +- 4.759	56.70 +- 7.244	1.55 +- 0.039	4.34 +- 4.582
16	97.72 +- 0.009	29.61 +- 10.057	64.86 +- 10.140	0.83 +- 0.107	2.42 +- 3.753
32	60.95 +- 0.052	10.51 +- 3.908	48.87 +- 4.319	0.41 +- 0.102	1.16 +- 2.212

BASE+WSHR-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	535.24 +- 0.000	504.70 +- 0.000	6.58 +- 0.000	8.57 +- 0.000	15.40 +- 0.000
2	356.72 +- 0.000	316.32 +- 12.178	26.42 +- 12.809	5.28 +- 0.229	8.71 +- 0.402
4	189.08 +- 0.001	145.49 +- 6.054	34.07 +- 6.020	3.07 +- 0.105	6.45 +- 2.237
8	178.09 +- 0.006	99.39 +- 8.323	72.56 +- 10.268	1.71 +- 0.070	4.43 +- 4.135
16	248.95 +- 0.010	67.98 +- 21.826	176.42 +- 21.851	1.00 +- 0.114	3.54 +- 7.633
32	106.46 +- 0.068	19.21 +- 5.232	85.63 +- 5.899	0.44 +- 0.068	1.18 +- 3.665

BASE+WSHR-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	515.89 +- 0.000	472.81 +- 0.000	8.58 +- 0.000	12.38 +- 0.000	22.12 +- 0.000
2	304.27 +- 0.000	253.58 +- 12.934	24.33 +- 6.691	8.96 +- 1.916	17.39 +- 4.327
4	193.95 +- 0.003	122.98 +- 10.289	50.16 +- 4.807	8.61 +- 1.070	12.19 +- 8.721
8	130.72 +- 0.011	55.24 +- 5.236	51.90 +- 17.036	5.22 +- 1.358	18.35 +- 14.142
16	409.39 +- 0.062	82.90 +- 9.920	206.63 +- 76.788	6.86 +- 3.136	113.00 +- 74.432
32	106.46 +- 0.068	19.21 +- 5.232	85.63 +- 5.899	0.44 +- 0.068	1.18 +- 3.664

BASE+WSTL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	520.41 +- 0.000	488.63 +- 0.000	7.32 +- 0.000	8.57 +- 0.000	15.89 +- 0.000
2	328.44 +- 0.001	244.05 +- 1.846	66.32 +- 2.245	23.31 +- 0.133	10.24 +- 0.532
4	226.85 +- 0.004	121.85 +- 23.331	83.87 +- 25.280	25.71 +- 10.007	14.58 +- 12.568
8	166.85 +- 0.007	61.23 +- 12.574	79.26 +- 15.862	23.27 +- 5.932	16.90 +- 10.421
16	124.39 +- 0.020	30.80 +- 10.584	79.93 +- 11.120	13.62 +- 2.756	9.97 +- 5.219
32	116.24 +- 0.032	15.72 +- 5.849	88.45 +- 6.172	0.38 +- 0.129	1.69 +- 0.678

BASE+WSTL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	546.73 +- 0.000	515.33 +- 0.000	7.36 +- 0.000	8.57 +- 0.000	15.47 +- 0.000
2	341.60 +- 0.001	255.15 +- 2.837	69.65 +- 1.906	23.79 +- 0.584	16.99 +- 0.349

Table B.5: (continued)

4	228.13 +- 0.004	127.88 +- 9.955	75.17 +- 10.429	25.15 +- 1.835	20.08 +- 6.856
8	156.84 +- 0.010	64.23 +- 21.008	88.76 +- 31.171	19.76 +- 7.162	25.90 +- 18.285
16	131.66 +- 0.026	32.26 +- 10.472	85.43 +- 12.304	13.48 +- 2.452	19.51 +- 7.626
32	118.67 +- 0.053	16.13 +- 6.170	100.53 +- 6.041	0.37 +- 0.139	1.64 +- 2.631

BASE+WSTL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	486.87 +- 0.000	442.54 +- 0.000	9.08 +- 0.000	12.38 +- 0.000	22.87 +- 0.000
2	306.59 +- 0.000	224.35 +- 4.547	60.13 +- 7.030	7.59 +- 0.896	14.52 +- 1.588
4	206.87 +- 0.003	112.26 +- 10.231	79.63 +- 13.951	5.39 +- 1.138	9.58 +- 2.851
8	156.56 +- 0.010	56.26 +- 13.349	91.29 +- 16.673	3.15 +- 1.307	5.85 +- 4.092
16	107.66 +- 0.012	28.08 +- 8.943	71.72 +- 12.041	1.74 +- 0.752	6.11 +- 4.366
32	86.32 +- 0.062	14.10 +- 5.207	65.98 +- 7.091	0.75 +- 0.377	5.50 +- 4.075

BASE+WSTLGBL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	530.86 +- 0.000	486.96 +- 0.000	9.77 +- 0.000	8.57 +- 0.000	25.56 +- 0.000
2	328.11 +- 0.000	243.87 +- 2.078	66.17 +- 2.487	23.31 +- 0.172	15.24 +- 0.581
4	214.69 +- 0.002	121.89 +- 15.128	81.88 +- 17.811	23.38 +- 8.680	12.46 +- 11.764
8	151.64 +- 0.012	61.18 +- 22.050	83.79 +- 23.226	16.64 +- 4.966	9.97 +- 8.807
16	123.89 +- 0.013	30.85 +- 10.806	89.19 +- 12.256	12.90 +- 2.939	9.06 +- 6.815
32	117.64 +- 0.036	15.47 +- 8.556	99.63 +- 9.845	10.99 +- 3.018	8.45 +- 4.078

BASE+WSTLGBL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	553.55 +- 0.000	509.74 +- 0.000	9.77 +- 0.000	8.57 +- 0.000	25.48 +- 0.000
2	341.66 +- 0.001	255.19 +- 2.815	69.67 +- 1.878	23.79 +- 0.578	17.00 +- 0.358
4	228.17 +- 0.004	127.90 +- 9.948	85.19 +- 10.418	25.16 +- 1.834	20.08 +- 6.854
8	172.60 +- 0.012	64.14 +- 15.664	99.04 +- 19.428	21.97 +- 6.194	25.54 +- 14.018
16	134.90 +- 0.018	32.33 +- 10.349	100.47 +- 13.562	14.27 +- 3.456	24.17 +- 10.400
32	117.72 +- 0.165	16.28 +- 7.751	97.19 +- 10.768	7.61 +- 1.337	16.36 +- 5.467

BASE+WSTLGBL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	517.44 +- 0.000	459.60 +- 0.000	10.04 +- 0.000	12.38 +- 0.000	35.41 +- 0.000
2	321.24 +- 0.002	230.59 +- 4.576	10.03 +- 3.702	25.35 +- 0.040	55.27 +- 0.832
4	234.33 +- 0.002	115.80 +- 30.765	32.01 +- 13.483	32.55 +- 10.077	53.97 +- 15.932
8	159.43 +- 0.006	57.81 +- 16.254	44.07 +- 8.490	22.79 +- 5.493	34.76 +- 10.963
16	100.50 +- 0.009	28.89 +- 9.274	31.17 +- 10.782	11.36 +- 2.537	29.08 +- 7.989
32	95.78 +- 0.077	14.50 +- 6.453	48.10 +- 11.645	6.64 +- 1.487	26.55 +- 9.559

Table B.5: (continued)

BASE+VC-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	524.24 +- 0.000	484.92 +- 0.000	8.40 +- 0.000	8.57 +- 0.000	22.35 +- 0.000
2	291.02 +- 0.001	243.19 +- 14.229	30.40 +- 14.639	5.06 +- 0.218	12.37 +- 0.192
4	148.00 +- 0.003	121.34 +- 5.836	17.50 +- 6.029	2.74 +- 0.100	6.42 +- 0.205
8	95.66 +- 0.003	61.28 +- 4.571	29.56 +- 4.619	1.44 +- 0.057	3.38 +- 0.172
16	84.06 +- 0.011	42.16 +- 11.665	38.18 +- 12.273	0.92 +- 0.104	2.80 +- 3.115
32	383.49 +- 0.035	39.23 +- 12.467	336.69 +- 34.378	0.81 +- 0.125	6.77 +- 27.402

BASE+VC-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	568.30 +- 0.000	528.12 +- 0.000	8.87 +- 0.000	8.57 +- 0.000	22.73 +- 0.000
2	309.53 +- 0.001	260.66 +- 11.024	31.31 +- 11.523	5.03 +- 0.222	12.53 +- 0.275
4	157.93 +- 0.007	129.72 +- 7.242	18.96 +- 7.392	2.71 +- 0.100	6.54 +- 0.187
8	101.39 +- 0.007	64.99 +- 5.115	31.59 +- 5.172	1.41 +- 0.049	3.40 +- 0.165
16	78.34 +- 0.007	46.62 +- 9.539	28.51 +- 9.795	0.94 +- 0.083	2.27 +- 0.862
32	108.05 +- 0.052	22.35 +- 9.774	83.40 +- 10.179	0.54 +- 0.133	1.75 +- 2.540

BASE+VC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	495.09 +- 0.000	452.13 +- 0.000	5.38 +- 0.000	12.38 +- 0.000	25.19 +- 0.000
2	243.22 +- 0.000	204.00 +- 11.899	18.17 +- 14.905	7.48 +- 0.889	13.57 +- 2.117
4	115.34 +- 0.001	94.42 +- 2.385	8.18 +- 4.710	4.38 +- 0.595	8.35 +- 1.942
8	95.14 +- 0.008	63.16 +- 4.895	24.61 +- 4.924	2.52 +- 0.584	4.84 +- 1.635
16	57.19 +- 0.027	27.06 +- 3.738	15.89 +- 6.435	2.02 +- 0.436	12.23 +- 4.602
32	270.60 +- 0.097	74.43 +- 16.105	98.59 +- 39.984	3.60 +- 0.706	93.99 +- 35.407

BASE+VC+WSHR-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	524.24 +- 0.000	484.92 +- 0.000	8.40 +- 0.000	8.57 +- 0.000	22.35 +- 0.000
2	291.02 +- 0.001	243.19 +- 14.229	30.40 +- 14.639	5.06 +- 0.218	12.37 +- 0.192
4	148.00 +- 0.003	121.34 +- 5.836	17.50 +- 6.029	2.74 +- 0.100	6.42 +- 0.205
8	95.66 +- 0.003	61.28 +- 4.571	29.56 +- 4.619	1.44 +- 0.057	3.38 +- 0.172
16	84.06 +- 0.011	42.16 +- 11.665	38.18 +- 12.273	0.92 +- 0.104	2.80 +- 3.115
32	383.49 +- 0.035	39.23 +- 12.467	336.69 +- 34.378	0.81 +- 0.125	6.77 +- 27.402

BASE+VC+WSHR-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	568.30 +- 0.000	528.12 +- 0.000	8.87 +- 0.000	8.57 +- 0.000	22.73 +- 0.000
2	309.53 +- 0.001	260.66 +- 11.024	31.31 +- 11.523	5.03 +- 0.222	12.53 +- 0.275
4	157.93 +- 0.007	129.72 +- 7.242	18.96 +- 7.392	2.71 +- 0.100	6.54 +- 0.187
8	101.39 +- 0.007	64.99 +- 5.115	31.59 +- 5.172	1.41 +- 0.049	3.40 +- 0.165
16	78.34 +- 0.007	46.62 +- 9.539	28.51 +- 9.795	0.94 +- 0.083	2.27 +- 0.862

Table B.5: (continued)

32	108.05 +- 0.052	22.35 +- 9.774	83.40 +- 10.179	0.54 +- 0.133	1.75 +- 2.540

BASE+VC+WSHR-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	495.09 +- 0.000	452.13 +- 0.000	5.38 +- 0.000	12.38 +- 0.000	25.19 +- 0.000
2	243.22 +- 0.000	204.00 +- 11.899	18.17 +- 14.905	7.48 +- 0.889	13.57 +- 2.117
4	115.34 +- 0.001	94.42 +- 2.385	8.18 +- 4.710	4.38 +- 0.595	8.35 +- 1.942
8	95.14 +- 0.008	63.16 +- 4.895	24.61 +- 4.924	2.52 +- 0.584	4.84 +- 1.635
16	57.19 +- 0.027	27.06 +- 3.738	15.89 +- 6.435	2.02 +- 0.436	12.23 +- 4.602
32	270.60 +- 0.097	74.43 +- 16.105	98.59 +- 39.984	3.60 +- 0.706	93.99 +- 35.407

BASE+VC+WSTL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	525.79 +- 0.000	491.91 +- 0.000	5.71 +- 0.000	8.57 +- 0.000	19.59 +- 0.000
2	330.67 +- 0.001	249.03 +- 4.541	64.93 +- 5.203	5.00 +- 0.299	11.72 +- 0.363
4	249.37 +- 0.001	125.79 +- 38.867	114.33 +- 42.122	2.70 +- 0.973	6.55 +- 2.329
8	160.57 +- 0.007	63.37 +- 26.608	92.32 +- 29.047	1.40 +- 0.749	3.48 +- 1.919
16	81.88 +- 0.019	31.25 +- 10.494	48.03 +- 11.494	0.74 +- 0.327	1.87 +- 0.811
32	59.95 +- 0.068	15.68 +- 4.878	42.68 +- 5.253	0.38 +- 0.138	1.21 +- 0.375

BASE+VC+WSTL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	555.68 +- 0.000	520.62 +- 0.000	6.36 +- 0.000	8.57 +- 0.000	20.12 +- 0.000
2	344.53 +- 0.000	260.78 +- 4.267	67.24 +- 4.948	4.99 +- 0.303	11.51 +- 0.379
4	238.70 +- 0.002	131.17 +- 30.523	98.58 +- 33.219	2.67 +- 0.809	6.29 +- 1.946
8	140.41 +- 0.005	65.55 +- 22.020	70.13 +- 24.070	1.39 +- 0.652	3.34 +- 1.538
16	93.83 +- 0.010	32.77 +- 7.999	58.59 +- 8.882	0.72 +- 0.300	1.76 +- 0.697
32	73.54 +- 0.069	16.45 +- 7.613	55.51 +- 8.008	0.36 +- 0.137	1.21 +- 0.364

BASE+VC+WSTL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	495.09 +- 0.000	452.13 +- 0.000	5.38 +- 0.000	12.38 +- 0.000	25.19 +- 0.000
2	363.39 +- 0.005	231.12 +- 95.466	111.20 +- 105.608	6.60 +- 3.230	14.47 +- 6.907
4	213.23 +- 0.005	114.45 +- 33.078	86.49 +- 37.055	3.81 +- 1.265	8.48 +- 2.729
8	131.34 +- 0.015	57.25 +- 10.647	66.60 +- 11.167	2.21 +- 0.564	5.28 +- 1.268
16	81.64 +- 0.055	28.95 +- 10.838	48.27 +- 11.546	1.24 +- 0.475	3.17 +- 1.266
32	57.40 +- 0.049	14.45 +- 5.125	40.68 +- 5.222	0.57 +- 0.201	1.70 +- 0.799

BASE+VC+WSTLGBL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	521.26 +- 0.000	488.70 +- 0.000	5.53 +- 0.000	8.57 +- 0.000	18.45 +- 0.000
2	276.33 +- 0.000	244.82 +- 5.899	15.25 +- 5.568	5.06 +- 0.092	11.20 +- 0.239
4	148.84 +- 0.003	122.95 +- 5.755	17.22 +- 5.684	2.75 +- 0.071	5.93 +- 0.134

Table B.5: (continued)

8	84.94 +- 0.006	61.58 +- 5.537	18.86 +- 5.575	1.44 +- 0.061	3.06 +- 0.164
16	56.61 +- 0.007	30.85 +- 4.886	23.37 +- 5.114	0.75 +- 0.079	1.64 +- 0.220
32	45.55 +- 0.068	15.45 +- 4.848	28.77 +- 4.998	0.39 +- 0.068	0.94 +- 0.139

BASE+VC+WSTLGBL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	559.56 +- 0.000	525.92 +- 0.000	6.14 +- 0.000	8.57 +- 0.000	18.93 +- 0.000
2	291.80 +- 0.000	262.32 +- 5.158	13.53 +- 4.739	5.03 +- 0.130	10.92 +- 0.289
4	159.04 +- 0.003	130.90 +- 7.153	19.58 +- 7.288	2.71 +- 0.015	5.84 +- 0.154
8	89.80 +- 0.004	65.46 +- 5.923	19.45 +- 6.120	1.41 +- 0.083	3.48 +- 0.247
16	56.45 +- 0.020	32.70 +- 5.851	21.42 +- 5.933	0.72 +- 0.059	1.60 +- 0.137
32	50.35 +- 0.059	16.43 +- 4.319	32.66 +- 4.466	0.37 +- 0.060	0.89 +- 0.128

BASE+VC+WSTLGBL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	496.75 +- 0.000	454.60 +- 0.000	4.51 +- 0.000	12.38 +- 0.000	25.26 +- 0.000
2	259.86 +- 0.000	227.88 +- 3.447	5.83 +- 1.062	8.02 +- 1.216	18.13 +- 3.294
4	137.92 +- 0.003	113.94 +- 1.939	8.70 +- 3.731	4.59 +- 0.715	10.70 +- 1.943
8	78.54 +- 0.005	57.17 +- 4.161	13.24 +- 4.420	2.37 +- 0.334	5.76 +- 0.875
16	49.85 +- 0.016	28.73 +- 4.509	17.11 +- 4.271	1.13 +- 0.264	2.88 +- 1.082
32	42.31 +- 0.026	14.36 +- 4.578	25.69 +- 4.620	0.55 +- 0.160	1.71 +- 1.743

Table B.6: Gröbner execution times (in seconds) on the SGI Origin 2000.

BASE-CC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	72.08 +- 0.000	71.68 +- 0.000	0.40 +- 0.000	0.00 +- 0.000	0.00 +- 0.000
2	37.06 +- 0.417	36.72 +- 0.290	0.33 +- 0.126	0.00 +- 0.000	0.01 +- 0.001
4	19.29 +- 0.084	18.80 +- 0.438	0.45 +- 0.436	0.00 +- 0.000	0.03 +- 0.004
8	11.01 +- 0.183	10.62 +- 0.166	0.34 +- 0.191	0.00 +- 0.000	0.05 +- 0.005
16	8.86 +- 0.115	8.39 +- 0.167	0.33 +- 0.152	0.00 +- 0.000	0.14 +- 0.011

BASE-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	73.98 +- 0.000	70.44 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000
2	37.51 +- 1.293	33.46 +- 0.464	2.26 +- 0.874	1.77 +- 0.045	0.01 +- 0.000
4	39.65 +- 0.913	30.94 +- 0.984	7.53 +- 1.545	1.17 +- 0.035	0.01 +- 0.002
8	25.74 +- 0.284	15.95 +- 0.797	9.13 +- 1.034	0.65 +- 0.036	0.01 +- 0.004
16	17.31 +- 0.601	8.28 +- 0.678	8.72 +- 0.946	0.31 +- 0.033	0.01 +- 0.004

Table B.6: (continued)

BASE-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	75.85 +- 0.000	72.31 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000
2	40.25 +- 0.015	35.46 +- 0.657	2.98 +- 0.650	1.80 +- 0.022	0.01 +- 0.000
4	30.80 +- 1.148	24.75 +- 1.176	4.97 +- 1.123	1.08 +- 0.033	0.01 +- 0.002
8	17.18 +- 0.282	9.77 +- 0.409	6.91 +- 0.565	0.49 +- 0.016	0.01 +- 0.003
16	18.55 +- 0.537	8.80 +- 0.597	9.39 +- 0.764	0.36 +- 0.036	0.01 +- 0.004

BASE-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	76.90 +- 0.000	72.41 +- 0.000	0.40 +- 0.000	4.10 +- 0.000	0.00 +- 0.000
2	45.94 +- 0.060	42.03 +- 0.013	1.16 +- 0.526	2.74 +- 0.479	0.00 +- 0.000
4	30.90 +- 0.918	25.32 +- 1.731	3.23 +- 2.092	2.34 +- 0.566	0.00 +- 0.000
8	33.03 +- 0.406	24.67 +- 2.009	6.57 +- 2.015	1.79 +- 0.586	0.00 +- 0.000
16	29.63 +- 0.715	20.80 +- 2.288	7.50 +- 2.591	1.33 +- 0.468	0.00 +- 0.000

BASE+WSHR-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	73.98 +- 0.000	70.44 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000
2	37.51 +- 1.293	33.46 +- 0.464	2.26 +- 0.874	1.77 +- 0.045	0.01 +- 0.000
4	39.65 +- 0.913	30.94 +- 0.984	7.53 +- 1.545	1.17 +- 0.035	0.01 +- 0.002
8	25.74 +- 0.284	15.95 +- 0.797	9.13 +- 1.034	0.65 +- 0.036	0.01 +- 0.004
16	17.31 +- 0.601	8.28 +- 0.678	8.72 +- 0.946	0.31 +- 0.033	0.01 +- 0.004

BASE+WSHR-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	75.85 +- 0.000	72.31 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000
2	40.25 +- 0.015	35.46 +- 0.657	2.98 +- 0.650	1.80 +- 0.022	0.01 +- 0.000
4	30.80 +- 1.148	24.75 +- 1.176	4.97 +- 1.123	1.08 +- 0.033	0.01 +- 0.002
8	17.18 +- 0.282	9.77 +- 0.409	6.91 +- 0.565	0.49 +- 0.016	0.01 +- 0.003
16	18.55 +- 0.537	8.80 +- 0.597	9.39 +- 0.764	0.36 +- 0.036	0.01 +- 0.004

BASE+WSHR-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	76.90 +- 0.000	72.41 +- 0.000	0.40 +- 0.000	4.10 +- 0.000	0.00 +- 0.000
2	45.94 +- 0.060	42.03 +- 0.013	1.16 +- 0.526	2.74 +- 0.479	0.00 +- 0.000
4	30.90 +- 0.918	25.32 +- 1.731	3.23 +- 2.092	2.34 +- 0.566	0.00 +- 0.000
8	33.03 +- 0.406	24.67 +- 2.009	6.57 +- 2.015	1.79 +- 0.586	0.00 +- 0.000
16	29.63 +- 0.715	20.80 +- 2.288	7.50 +- 2.591	1.33 +- 0.468	0.00 +- 0.000

BASE+WSTL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	74.93 +- 0.000	71.39 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000

Table B.6: (continued)

2	46.72 +- 0.050	31.08 +- 0.301	2.84 +- 0.146	12.77 +- 0.396	0.02 +- 0.001
4	31.98 +- 0.411	10.98 +- 2.396	4.88 +- 0.489	16.11 +- 2.230	0.02 +- 0.004
8	23.32 +- 0.412	3.66 +- 3.635	7.18 +- 0.774	12.44 +- 3.346	0.04 +- 0.007
16	17.71 +- 0.488	2.93 +- 1.816	9.04 +- 1.507	5.66 +- 1.097	0.08 +- 0.008

BASE+WSTL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	74.34 +- 0.000	70.80 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000
2	46.59 +- 0.072	30.87 +- 0.413	2.88 +- 0.193	12.82 +- 0.533	0.02 +- 0.001
4	30.45 +- 0.200	12.42 +- 6.235	5.33 +- 0.940	12.67 +- 5.433	0.02 +- 0.003
8	20.83 +- 0.141	5.12 +- 2.437	6.39 +- 1.045	9.28 +- 1.925	0.04 +- 0.006
16	17.06 +- 0.468	2.04 +- 2.123	9.15 +- 1.687	5.80 +- 1.194	0.06 +- 0.007

BASE+WSTL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	79.03 +- 0.000	74.54 +- 0.000	0.40 +- 0.000	4.10 +- 0.000	0.00 +- 0.000
2	59.06 +- 0.636	32.63 +- 20.106	3.01 +- 1.868	22.91 +- 18.356	0.52 +- 0.518
4	34.83 +- 0.305	20.91 +- 3.400	1.84 +- 0.270	11.84 +- 3.343	0.24 +- 0.172
8	22.33 +- 0.180	4.66 +- 3.525	7.37 +- 1.086	10.25 +- 2.904	0.05 +- 0.006
16	20.29 +- 0.485	11.85 +- 1.598	2.70 +- 1.075	5.31 +- 1.211	0.43 +- 0.184

BASE+WSTLGBL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	74.96 +- 0.000	71.41 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000
2	39.78 +- 0.007	35.49 +- 0.050	2.45 +- 0.037	1.82 +- 0.021	0.01 +- 0.001
4	23.74 +- 0.076	17.99 +- 0.439	4.41 +- 0.346	1.32 +- 0.196	0.02 +- 0.007
8	15.76 +- 0.151	9.31 +- 0.996	5.69 +- 1.086	0.75 +- 0.197	0.01 +- 0.009
16	13.56 +- 0.285	4.91 +- 1.380	8.30 +- 1.471	0.34 +- 0.109	0.01 +- 0.011

BASE+WSTLGBL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	74.44 +- 0.000	70.90 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000
2	39.65 +- 0.012	35.37 +- 0.180	2.45 +- 0.141	1.81 +- 0.029	0.01 +- 0.001
4	23.03 +- 0.083	17.88 +- 0.356	4.04 +- 0.498	1.09 +- 0.129	0.01 +- 0.006
8	15.52 +- 0.293	9.16 +- 0.620	5.59 +- 0.601	0.76 +- 0.209	0.01 +- 0.008
16	13.14 +- 0.352	4.63 +- 1.099	8.21 +- 1.198	0.30 +- 0.081	0.01 +- 0.010

BASE+WSTLGBL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	75.67 +- 0.000	71.18 +- 0.000	0.40 +- 0.000	4.10 +- 0.000	0.00 +- 0.000
2	40.95 +- 0.060	36.49 +- 1.308	1.09 +- 0.162	3.35 +- 1.102	0.02 +- 0.015
4	24.61 +- 0.741	20.70 +- 0.675	1.37 +- 0.827	2.31 +- 0.597	0.23 +- 0.196
8	17.22 +- 0.107	14.15 +- 0.269	1.46 +- 0.426	1.37 +- 0.215	0.23 +- 0.161

Table B.6: (continued)

16	14.67 +- 0.407	11.04 +- 1.640	2.26 +- 1.409	0.96 +- 0.269	0.40 +- 0.279

BASE+VC-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	73.51 +- 0.000	69.97 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000
2	43.26 +- 0.193	38.34 +- 0.754	3.06 +- 1.026	1.86 +- 0.080	0.01 +- 0.000
4	25.66 +- 0.113	21.85 +- 0.733	2.74 +- 0.908	1.06 +- 0.073	0.01 +- 0.003
8	23.41 +- 0.325	17.46 +- 1.536	5.34 +- 1.881	0.60 +- 0.049	0.01 +- 0.005
16	14.61 +- 0.611	9.72 +- 0.894	4.45 +- 0.955	0.43 +- 0.041	0.01 +- 0.006

BASE+VC-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	73.55 +- 0.000	70.01 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000
2	43.50 +- 0.035	40.44 +- 0.508	1.24 +- 0.430	1.82 +- 0.043	0.01 +- 0.000
4	21.12 +- 0.083	18.46 +- 0.414	1.71 +- 0.458	0.94 +- 0.028	0.01 +- 0.002
8	15.68 +- 0.233	12.45 +- 0.715	2.66 +- 0.889	0.57 +- 0.030	0.01 +- 0.003
16	11.48 +- 0.595	7.14 +- 0.847	4.04 +- 1.256	0.30 +- 0.034	0.01 +- 0.003

BASE+VC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	75.01 +- 0.000	70.51 +- 0.000	0.40 +- 0.000	4.10 +- 0.000	0.00 +- 0.000
2	50.67 +- 0.077	47.04 +- 0.985	0.94 +- 0.714	2.69 +- 0.349	0.00 +- 0.000
4	28.83 +- 0.374	25.76 +- 0.619	1.46 +- 0.839	1.61 +- 0.126	0.00 +- 0.000
8	20.25 +- 0.247	17.00 +- 1.249	2.09 +- 1.377	1.15 +- 0.261	0.00 +- 0.000
16	44.99 +- 5.838	30.71 +- 1.713	13.04 +- 5.706	1.24 +- 1.108	0.00 +- 0.000

BASE+VC+WSHR-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	73.51 +- 0.000	69.97 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000
2	43.26 +- 0.193	38.34 +- 0.754	3.06 +- 1.026	1.86 +- 0.080	0.01 +- 0.000
4	25.66 +- 0.113	21.85 +- 0.733	2.74 +- 0.908	1.06 +- 0.073	0.01 +- 0.003
8	23.41 +- 0.325	17.46 +- 1.536	5.34 +- 1.881	0.60 +- 0.049	0.01 +- 0.005
16	14.61 +- 0.611	9.72 +- 0.894	4.45 +- 0.955	0.43 +- 0.041	0.01 +- 0.006

BASE+VC+WSHR-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	73.55 +- 0.000	70.01 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000
2	43.50 +- 0.035	40.44 +- 0.508	1.24 +- 0.430	1.82 +- 0.043	0.01 +- 0.000
4	21.12 +- 0.083	18.46 +- 0.414	1.71 +- 0.458	0.94 +- 0.028	0.01 +- 0.002
8	15.68 +- 0.233	12.45 +- 0.715	2.66 +- 0.889	0.57 +- 0.030	0.01 +- 0.003
16	11.48 +- 0.595	7.14 +- 0.847	4.04 +- 1.256	0.30 +- 0.034	0.01 +- 0.003

Table B.6: (continued)

BASE+VC+WSHR-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	75.01 +- 0.000	70.51 +- 0.000	0.40 +- 0.000	4.10 +- 0.000	0.00 +- 0.000
2	50.67 +- 0.077	47.04 +- 0.985	0.94 +- 0.714	2.69 +- 0.349	0.00 +- 0.000
4	28.83 +- 0.374	25.76 +- 0.619	1.46 +- 0.839	1.61 +- 0.126	0.00 +- 0.000
8	20.25 +- 0.247	17.00 +- 1.249	2.09 +- 1.377	1.15 +- 0.261	0.00 +- 0.000
16	44.99 +- 5.838	30.71 +- 1.713	13.04 +- 5.706	1.24 +- 1.108	0.00 +- 0.000

BASE+VC+WSTL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	75.13 +- 0.000	71.59 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000
2	46.34 +- 0.069	39.48 +- 0.156	5.06 +- 0.150	1.79 +- 0.064	0.02 +- 0.001
4	32.95 +- 0.680	24.29 +- 2.611	7.69 +- 3.586	0.95 +- 0.317	0.02 +- 0.003
8	22.60 +- 0.335	15.02 +- 1.622	7.05 +- 2.144	0.49 +- 0.243	0.03 +- 0.005
16	12.57 +- 0.514	8.19 +- 0.668	4.07 +- 1.023	0.26 +- 0.112	0.05 +- 0.004

BASE+VC+WSTL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	75.20 +- 0.000	71.65 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000
2	46.25 +- 0.077	39.37 +- 0.165	5.09 +- 0.153	1.78 +- 0.066	0.02 +- 0.001
4	31.94 +- 0.411	24.05 +- 2.729	6.90 +- 2.995	0.94 +- 0.311	0.05 +- 0.002
8	18.72 +- 0.291	13.23 +- 1.590	4.98 +- 2.026	0.48 +- 0.206	0.04 +- 0.003
16	13.41 +- 0.516	8.66 +- 0.793	4.45 +- 1.131	0.25 +- 0.107	0.05 +- 0.002

BASE+VC+WSTL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	77.60 +- 0.000	73.10 +- 0.000	0.40 +- 0.000	4.10 +- 0.000	0.00 +- 0.000
2	55.76 +- 0.593	44.30 +- 6.216	8.68 +- 7.160	2.22 +- 0.904	0.56 +- 0.553
4	31.24 +- 0.508	23.88 +- 1.122	5.93 +- 0.778	1.23 +- 0.231	0.19 +- 0.113
8	22.78 +- 0.543	15.81 +- 1.571	6.10 +- 2.201	0.67 +- 0.308	0.20 +- 0.129
16	13.00 +- 0.551	9.30 +- 0.527	2.74 +- 0.739	0.32 +- 0.105	0.65 +- 0.188

BASE+VC+WSTLGBL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	75.05 +- 0.000	71.51 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000
2	39.29 +- 0.010	35.93 +- 0.940	1.55 +- 0.924	1.80 +- 0.007	0.02 +- 0.001
4	21.33 +- 0.327	18.70 +- 0.312	1.65 +- 0.566	0.97 +- 0.035	0.02 +- 0.007
8	12.11 +- 0.298	10.00 +- 0.477	1.59 +- 0.485	0.51 +- 0.039	0.01 +- 0.010
16	7.91 +- 0.325	5.89 +- 0.343	1.75 +- 0.424	0.26 +- 0.031	0.01 +- 0.008

BASE+VC+WSTLGBL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	74.95 +- 0.000	71.41 +- 0.000	0.40 +- 0.000	3.14 +- 0.000	0.00 +- 0.000

Table B.6: (continued)

2	40.26 +- 0.210	36.89 +- 0.802	1.56 +- 0.948	1.79 +- 0.066	0.01 +- 0.001
4	21.29 +- 0.480	18.79 +- 0.792	1.54 +- 0.363	0.95 +- 0.040	0.01 +- 0.006
8	12.06 +- 0.125	10.04 +- 0.350	1.52 +- 0.458	0.49 +- 0.025	0.01 +- 0.006
16	7.83 +- 0.419	5.83 +- 0.410	1.74 +- 0.431	0.25 +- 0.021	0.01 +- 0.006

BASE+VC+WSTLGBL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	77.68 +- 0.000	73.18 +- 0.000	0.40 +- 0.000	4.10 +- 0.000	0.00 +- 0.000
2	40.14 +- 0.010	36.72 +- 0.581	0.64 +- 0.191	2.77 +- 0.375	0.01 +- 0.005
4	22.34 +- 0.325	20.36 +- 0.334	0.51 +- 0.079	1.34 +- 0.172	0.13 +- 0.108
8	12.19 +- 0.135	10.54 +- 0.354	0.57 +- 0.151	0.76 +- 0.188	0.32 +- 0.321
16	7.63 +- 0.419	5.62 +- 0.408	1.74 +- 0.431	0.26 +- 0.024	0.01 +- 0.006

B.4 Phylogeny

Tables B.7 and B.8 list the execution times for the phylogeny application on the Cray T3D and the SGI Origin 2000, respectively.

Measurements are presented for different numbers of processors, across the range of communication architectures, and for different levels of support of data locality and load balancing. The prefix of the configuration label denotes support for data locality and load balance: BASE, BASE+WSHR, BASE+WSTL, BASE+WSTLGBL, BASE+VC, BASE+VC+WSHR, BASE+VC+WSTL, and BASE+VC+WSTLGBL correspond to the baseline version of the application, and its augmentation with various load balancing policies and custom view coherence protocols described in Section 7.5. The TOTAL, COMPUTE, IDLE, COHERENCE, and OVERHEAD columns show the total execution time, and its breakdown into compute time, idle time, coherence time, and runtime overhead components. The data in the tables are displayed graphically in Figures 7.21–7.24.

Table B.7: Phylogeny execution times (in seconds) on the Cray T3D.

BASE-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	260.37 +- 0.000	167.37 +- 0.000	26.22 +- 0.000	64.73 +- 0.000	2.04 +- 0.000
2	182.52 +- 0.000	79.07 +- 9.285	45.87 +- 9.650	52.12 +- 0.147	5.46 +- 0.512
4	97.62 +- 0.000	36.69 +- 5.932	24.98 +- 1.370	31.70 +- 6.050	4.25 +- 1.220

Table B.7: (continued)

8	64.73 +- 0.000	11.31 +- 14.384	26.16 +- 4.638	22.79 +- 15.807	4.47 +- 3.207
16	71.60 +- 0.000	5.23 +- 20.483	44.89 +- 8.481	17.46 +- 23.833	4.02 +- 5.018
32	87.82 +- 0.000	6.69 +- 24.696	66.61 +- 8.656	11.58 +- 26.311	2.94 +- 5.699

BASE-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	256.86 +- 0.000	164.07 +- 0.000	26.02 +- 0.000	64.73 +- 0.000	2.04 +- 0.000
2	182.10 +- 0.000	87.27 +- 8.060	43.82 +- 9.032	45.92 +- 0.986	5.09 +- 0.014
4	91.54 +- 0.000	40.10 +- 3.476	22.14 +- 0.257	25.71 +- 1.809	3.58 +- 1.474
8	62.18 +- 0.000	17.58 +- 8.240	23.89 +- 3.875	16.57 +- 8.041	4.14 +- 4.063
16	59.86 +- 0.000	7.99 +- 13.990	36.44 +- 6.240	11.41 +- 13.385	4.02 +- 6.747
32	68.73 +- 0.000	5.35 +- 14.630	53.52 +- 6.449	7.10 +- 13.747	2.76 +- 7.036

BASE-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	279.06 +- 0.000	166.96 +- 0.000	15.73 +- 0.000	94.32 +- 0.000	2.04 +- 0.000
2	174.77 +- 0.000	86.06 +- 7.633	24.12 +- 8.765	62.85 +- 1.150	1.74 +- 0.018
4	87.89 +- 0.000	38.94 +- 2.362	13.11 +- 0.720	34.81 +- 1.725	1.03 +- 0.004
8	64.14 +- 0.000	16.00 +- 5.344	23.41 +- 4.877	24.16 +- 9.765	0.56 +- 0.005
16	72.22 +- 0.000	24.64 +- 15.582	29.83 +- 11.558	17.46 +- 16.903	0.29 +- 0.005
32	94.51 +- 0.000	55.21 +- 19.298	28.62 +- 12.067	10.54 +- 17.873	0.15 +- 0.004

BASE+WSHR-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	260.37 +- 0.000	167.37 +- 0.000	26.22 +- 0.000	64.73 +- 0.000	2.04 +- 0.000
2	182.52 +- 0.000	79.07 +- 9.285	45.87 +- 9.650	52.12 +- 0.147	5.46 +- 0.512
4	97.62 +- 0.000	36.69 +- 5.932	24.98 +- 1.370	31.70 +- 6.050	4.25 +- 1.220
8	64.73 +- 0.000	11.31 +- 14.384	26.16 +- 4.638	22.79 +- 15.807	4.47 +- 3.207
16	71.60 +- 0.000	5.23 +- 20.483	44.89 +- 8.481	17.46 +- 23.833	4.02 +- 5.018
32	87.82 +- 0.000	6.69 +- 24.696	66.61 +- 8.656	11.58 +- 26.311	2.94 +- 5.699

BASE+WSHR-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	256.86 +- 0.000	164.07 +- 0.000	26.02 +- 0.000	64.73 +- 0.000	2.04 +- 0.000
2	182.10 +- 0.000	87.27 +- 8.060	43.82 +- 9.032	45.92 +- 0.986	5.09 +- 0.014
4	91.54 +- 0.000	40.10 +- 3.476	22.14 +- 0.257	25.71 +- 1.809	3.58 +- 1.474
8	62.18 +- 0.000	17.58 +- 8.240	23.89 +- 3.875	16.57 +- 8.041	4.14 +- 4.063
16	59.86 +- 0.000	7.99 +- 13.990	36.44 +- 6.240	11.41 +- 13.385	4.02 +- 6.747
32	68.73 +- 0.000	5.35 +- 14.630	53.52 +- 6.449	7.10 +- 13.747	2.76 +- 7.036

BASE+WSHR-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	279.06 +- 0.000	166.96 +- 0.000	15.73 +- 0.000	94.32 +- 0.000	2.04 +- 0.000

Table B.7: (continued)

2	174.77 +- 0.000	86.06 +- 7.633	24.12 +- 8.765	62.85 +- 1.150	1.74 +- 0.018
4	87.89 +- 0.000	38.94 +- 2.362	13.11 +- 0.720	34.81 +- 1.725	1.03 +- 0.004
8	64.14 +- 0.000	16.00 +- 5.344	23.41 +- 4.877	24.16 +- 9.765	0.56 +- 0.005
16	72.22 +- 0.000	24.64 +- 15.582	29.83 +- 11.558	17.46 +- 16.903	0.29 +- 0.005
32	94.51 +- 0.000	55.21 +- 19.298	28.62 +- 12.067	10.54 +- 17.873	0.15 +- 0.004

BASE+WSTL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	263.90 +- 0.000	170.16 +- 0.000	26.97 +- 0.000	64.73 +- 0.000	2.04 +- 0.000
2	139.69 +- 0.000	77.44 +- 4.158	20.44 +- 1.819	39.88 +- 2.285	1.93 +- 0.053
4	80.89 +- 0.000	40.06 +- 1.885	16.02 +- 2.892	23.11 +- 1.389	1.70 +- 0.426
8	46.05 +- 0.000	16.90 +- 3.652	13.54 +- 2.359	14.03 +- 4.699	1.59 +- 0.985
16	34.09 +- 0.000	8.79 +- 6.610	15.32 +- 1.843	8.61 +- 6.636	1.37 +- 1.432
32	37.20 +- 0.000	5.08 +- 10.066	24.78 +- 2.602	6.05 +- 9.877	1.29 +- 2.165

BASE+WSTL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	260.91 +- 0.000	167.30 +- 0.000	26.84 +- 0.000	64.73 +- 0.000	2.04 +- 0.000
2	140.61 +- 0.000	80.10 +- 6.545	20.65 +- 5.490	38.03 +- 1.132	1.83 +- 0.077
4	79.59 +- 0.000	41.50 +- 0.616	15.14 +- 1.518	21.37 +- 1.459	1.58 +- 0.586
8	47.33 +- 0.000	19.60 +- 1.668	14.31 +- 3.149	11.93 +- 2.433	1.49 +- 1.158
16	33.54 +- 0.000	9.38 +- 4.320	15.62 +- 2.619	7.09 +- 4.150	1.46 +- 2.104
32	29.95 +- 0.000	5.34 +- 5.497	19.38 +- 1.989	4.14 +- 4.608	1.08 +- 2.416

BASE+WSTL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	277.07 +- 0.000	164.76 +- 0.000	15.94 +- 0.000	94.32 +- 0.000	2.04 +- 0.000
2	147.83 +- 0.000	82.04 +- 4.146	10.50 +- 2.274	54.17 +- 1.916	1.12 +- 0.045
4	80.58 +- 0.000	39.49 +- 1.469	10.27 +- 3.037	30.14 +- 1.797	0.68 +- 0.077
8	48.82 +- 0.000	19.17 +- 1.580	12.46 +- 3.760	16.74 +- 3.130	0.45 +- 0.067
16	34.50 +- 0.000	11.96 +- 3.188	12.91 +- 3.061	9.29 +- 3.582	0.34 +- 0.070
32	33.11 +- 0.000	14.93 +- 4.492	12.54 +- 3.031	5.45 +- 4.635	0.19 +- 0.041

BASE+WSTLGBL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	256.99 +- 0.000	163.65 +- 0.000	26.21 +- 0.000	64.73 +- 0.000	2.40 +- 0.000
2	163.09 +- 0.000	81.02 +- 17.649	39.40 +- 22.773	38.68 +- 4.674	3.98 +- 0.450
4	98.79 +- 0.000	38.75 +- 8.677	33.04 +- 11.101	23.41 +- 1.894	3.58 +- 1.231
8	57.64 +- 0.000	17.12 +- 5.739	23.89 +- 5.359	14.02 +- 4.514	2.60 +- 1.557
16	38.94 +- 0.000	8.54 +- 6.153	20.14 +- 3.131	8.56 +- 5.986	1.70 +- 1.638
32	34.97 +- 0.000	4.73 +- 7.173	24.00 +- 2.748	5.09 +- 7.103	1.15 +- 1.741

Table B.7: (continued)

BASE+WSTLGBL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	257.43 +- 0.000	163.98 +- 0.000	26.32 +- 0.000	64.73 +- 0.000	2.40 +- 0.000
2	165.61 +- 0.000	84.02 +- 18.806	40.47 +- 23.723	37.28 +- 4.364	3.85 +- 0.553
4	98.71 +- 0.000	40.95 +- 8.251	32.62 +- 11.067	21.62 +- 1.696	3.51 +- 1.264
8	57.00 +- 0.000	19.72 +- 4.591	23.22 +- 5.596	11.78 +- 1.687	2.29 +- 1.561
16	38.74 +- 0.000	9.60 +- 4.465	20.57 +- 3.295	6.85 +- 3.391	1.72 +- 2.028
32	32.85 +- 0.000	5.49 +- 3.910	22.84 +- 2.725	3.57 +- 2.990	0.95 +- 1.718

BASE+WSTLGBL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	283.44 +- 0.000	170.88 +- 0.000	15.84 +- 0.000	94.32 +- 0.000	2.40 +- 0.000
2	175.53 +- 0.000	85.27 +- 16.284	33.77 +- 24.141	53.22 +- 7.308	3.26 +- 0.549
4	103.86 +- 0.000	41.95 +- 8.611	29.04 +- 12.597	30.44 +- 2.835	2.43 +- 1.256
8	61.59 +- 0.000	19.73 +- 4.864	23.74 +- 6.456	16.74 +- 2.574	1.38 +- 1.294
16	39.78 +- 0.000	12.26 +- 2.913	18.00 +- 3.185	8.80 +- 2.827	0.73 +- 1.067
32	35.17 +- 0.000	12.06 +- 3.076	18.09 +- 2.454	4.64 +- 2.620	0.37 +- 0.801

BASE+VC-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	232.79 +- 0.000	180.48 +- 0.000	20.96 +- 0.000	29.31 +- 0.000	2.04 +- 0.000
2	184.01 +- 0.000	97.26 +- 7.866	48.77 +- 11.375	32.01 +- 2.107	5.97 +- 1.403
4	73.88 +- 0.000	45.33 +- 0.727	14.75 +- 1.732	11.78 +- 1.888	2.03 +- 0.121
8	45.12 +- 0.000	22.09 +- 1.909	12.69 +- 3.009	8.75 +- 4.472	1.59 +- 0.420
16	43.90 +- 0.000	12.32 +- 0.805	25.48 +- 5.816	5.18 +- 4.618	0.92 +- 0.441
32	27.13 +- 0.000	7.27 +- 1.025	16.94 +- 2.724	2.47 +- 3.375	0.45 +- 0.348

BASE+VC-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	233.19 +- 0.000	180.64 +- 0.000	21.19 +- 0.000	29.31 +- 0.000	2.04 +- 0.000
2	134.29 +- 0.000	98.19 +- 0.789	16.62 +- 1.864	17.74 +- 2.656	1.73 +- 0.003
4	65.33 +- 0.000	48.21 +- 0.772	6.75 +- 0.112	9.34 +- 0.884	1.03 +- 0.005
8	34.77 +- 0.000	24.41 +- 0.545	4.24 +- 0.383	5.56 +- 0.513	0.56 +- 0.006
16	20.79 +- 0.000	13.50 +- 0.572	3.32 +- 0.302	3.67 +- 0.356	0.29 +- 0.007
32	9.90 +- 0.000	6.14 +- 0.268	2.25 +- 0.234	1.37 +- 0.069	0.15 +- 0.003

BASE+VC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	233.17 +- 0.000	180.56 +- 0.000	21.26 +- 0.000	29.31 +- 0.000	2.04 +- 0.000
2	130.61 +- 0.000	94.35 +- 2.502	15.14 +- 1.542	19.40 +- 4.049	1.72 +- 0.005
4	64.88 +- 0.000	47.13 +- 1.016	6.90 +- 0.387	9.82 +- 1.159	1.03 +- 0.005
8	34.61 +- 0.000	23.46 +- 0.889	4.13 +- 0.330	6.46 +- 0.853	0.56 +- 0.006
16	19.96 +- 0.000	12.46 +- 0.683	2.81 +- 0.247	4.40 +- 0.554	0.29 +- 0.007

Table B.7: (continued)

32	10.90 +- 0.000	5.91 +- 0.454	3.12 +- 0.347	1.72 +- 0.144	0.15 +- 0.003

BASE+VC+WSHR-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	232.79 +- 0.000	180.48 +- 0.000	20.96 +- 0.000	29.31 +- 0.000	2.04 +- 0.000
2	184.01 +- 0.000	97.26 +- 7.866	48.77 +- 11.375	32.01 +- 2.107	5.97 +- 1.403
4	73.88 +- 0.000	45.33 +- 0.727	14.75 +- 1.732	11.78 +- 1.888	2.03 +- 0.121
8	45.12 +- 0.000	22.09 +- 1.909	12.69 +- 3.009	8.75 +- 4.472	1.59 +- 0.420
16	43.90 +- 0.000	12.32 +- 0.805	25.48 +- 5.816	5.18 +- 4.618	0.92 +- 0.441
32	27.13 +- 0.000	7.27 +- 1.025	16.94 +- 2.724	2.47 +- 3.375	0.45 +- 0.348

BASE+VC+WSHR-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	233.19 +- 0.000	180.64 +- 0.000	21.19 +- 0.000	29.31 +- 0.000	2.04 +- 0.000
2	134.29 +- 0.000	98.19 +- 0.789	16.62 +- 1.864	17.74 +- 2.656	1.73 +- 0.003
4	65.33 +- 0.000	48.21 +- 0.772	6.75 +- 0.112	9.34 +- 0.884	1.03 +- 0.005
8	34.77 +- 0.000	24.41 +- 0.545	4.24 +- 0.383	5.56 +- 0.513	0.56 +- 0.006
16	20.79 +- 0.000	13.50 +- 0.572	3.32 +- 0.302	3.67 +- 0.356	0.29 +- 0.007
32	9.90 +- 0.000	6.14 +- 0.268	2.25 +- 0.234	1.37 +- 0.069	0.15 +- 0.003

BASE+VC+WSHR-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	233.17 +- 0.000	180.56 +- 0.000	21.26 +- 0.000	29.31 +- 0.000	2.04 +- 0.000
2	130.61 +- 0.000	94.35 +- 2.502	15.14 +- 1.542	19.40 +- 4.049	1.72 +- 0.005
4	64.88 +- 0.000	47.13 +- 1.016	6.90 +- 0.387	9.82 +- 1.159	1.03 +- 0.005
8	34.61 +- 0.000	23.46 +- 0.889	4.13 +- 0.330	6.46 +- 0.853	0.56 +- 0.006
16	19.96 +- 0.000	12.46 +- 0.683	2.81 +- 0.247	4.40 +- 0.554	0.29 +- 0.007
32	10.90 +- 0.000	5.91 +- 0.454	3.12 +- 0.347	1.72 +- 0.144	0.15 +- 0.003

BASE+VC+WSTL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	232.79 +- 0.000	180.68 +- 0.000	20.76 +- 0.000	29.31 +- 0.000	2.04 +- 0.000
2	122.81 +- 0.000	89.98 +- 4.036	15.90 +- 5.043	15.53 +- 1.019	1.41 +- 0.013
4	66.95 +- 0.000	44.82 +- 1.040	12.43 +- 1.779	8.69 +- 0.806	1.01 +- 0.140
8	37.37 +- 0.000	22.20 +- 0.940	9.56 +- 1.917	4.89 +- 1.137	0.72 +- 0.137
16	23.61 +- 0.000	11.14 +- 1.600	8.46 +- 1.285	3.40 +- 1.700	0.61 +- 0.226
32	13.00 +- 0.000	5.85 +- 1.253	5.41 +- 0.977	1.43 +- 1.094	0.31 +- 0.146

BASE+VC+WSTL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	237.97 +- 0.000	185.83 +- 0.000	20.79 +- 0.000	29.31 +- 0.000	2.04 +- 0.000
2	123.65 +- 0.000	93.59 +- 1.290	14.00 +- 2.827	14.87 +- 1.416	1.19 +- 0.122
4	69.32 +- 0.000	48.71 +- 2.331	12.07 +- 3.288	7.74 +- 0.991	0.80 +- 0.067

Table B.7: (continued)

8	35.15 +- 0.000	24.08 +- 1.334	6.49 +- 1.784	4.17 +- 0.576	0.42 +- 0.055
16	21.92 +- 0.000	12.48 +- 1.595	6.84 +- 2.002	2.26 +- 0.456	0.35 +- 0.063
32	11.27 +- 0.000	6.33 +- 0.763	3.64 +- 0.943	1.12 +- 0.199	0.18 +- 0.028

BASE+VC+WSTL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	248.38 +- 0.000	195.13 +- 0.000	20.93 +- 0.000	29.31 +- 0.000	3.00 +- 0.000
2	125.98 +- 0.000	98.58 +- 0.252	10.63 +- 0.054	15.01 +- 0.249	1.76 +- 0.057
4	65.67 +- 0.000	49.17 +- 0.997	7.05 +- 0.261	8.01 +- 0.738	1.44 +- 0.034
8	32.38 +- 0.000	24.60 +- 0.280	2.96 +- 0.049	4.39 +- 0.269	0.43 +- 0.012
16	16.44 +- 0.000	12.28 +- 0.206	1.60 +- 0.064	2.33 +- 0.164	0.22 +- 0.011
32	8.75 +- 0.000	6.34 +- 0.116	0.97 +- 0.052	1.32 +- 0.106	0.13 +- 0.010

BASE+VC+WSTLGBL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	241.28 +- 0.000	188.79 +- 0.000	20.78 +- 0.000	29.31 +- 0.000	2.40 +- 0.000
2	149.24 +- 0.000	92.24 +- 21.991	37.31 +- 24.891	16.00 +- 2.531	3.69 +- 0.369
4	87.41 +- 0.000	46.28 +- 8.583	29.15 +- 10.581	9.16 +- 1.016	2.81 +- 1.285
8	49.63 +- 0.000	22.27 +- 4.019	20.03 +- 5.313	5.58 +- 1.497	1.76 +- 1.318
16	29.69 +- 0.000	11.02 +- 2.180	14.04 +- 2.745	3.56 +- 2.278	1.07 +- 1.113
32	21.73 +- 0.000	5.76 +- 2.004	13.49 +- 2.224	1.91 +- 2.091	0.57 +- 0.832

BASE+VC+WSTLGBL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	245.46 +- 0.000	192.90 +- 0.000	20.84 +- 0.000	29.31 +- 0.000	2.40 +- 0.000
2	148.57 +- 0.000	96.06 +- 18.874	34.05 +- 21.783	15.07 +- 2.431	3.39 +- 0.478
4	87.20 +- 0.000	49.46 +- 9.549	27.40 +- 11.880	7.93 +- 1.094	2.41 +- 1.253
8	49.52 +- 0.000	25.25 +- 3.961	18.33 +- 5.545	4.56 +- 0.321	1.38 +- 1.306
16	29.11 +- 0.000	12.66 +- 1.967	13.29 +- 3.001	2.43 +- 0.088	0.73 +- 1.074
32	21.39 +- 0.000	6.52 +- 1.548	13.27 +- 2.273	1.23 +- 0.087	0.37 +- 0.803

BASE+VC+WSTLGBL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	245.45 +- 0.000	192.93 +- 0.000	20.81 +- 0.000	29.31 +- 0.000	2.40 +- 0.000
2	147.50 +- 0.000	95.61 +- 18.447	33.18 +- 21.009	15.32 +- 2.081	3.39 +- 0.481
4	87.06 +- 0.000	49.17 +- 8.899	26.80 +- 11.473	8.67 +- 1.396	2.42 +- 1.260
8	48.51 +- 0.000	24.64 +- 3.875	17.89 +- 5.439	4.60 +- 0.380	1.38 +- 1.301
16	28.71 +- 0.000	12.33 +- 1.911	13.09 +- 2.950	2.57 +- 0.147	0.73 +- 1.071
32	21.25 +- 0.000	6.34 +- 1.538	13.21 +- 2.263	1.33 +- 0.102	0.37 +- 0.802

Table B.8: Phylogeny execution times (in seconds) on the SGI Origin 2000.

BASE-CC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	39.03 +- 0.000	38.11 +- 0.000	0.00 +- 0.000	0.00 +- 0.000	0.92 +- 0.000
2	21.33 +- 0.033	20.86 +- 0.012	0.00 +- 0.000	0.00 +- 0.000	0.47 +- 0.022
4	13.22 +- 0.078	12.93 +- 0.092	0.00 +- 0.000	0.00 +- 0.000	0.30 +- 0.017
8	10.39 +- 0.059	10.07 +- 0.051	0.00 +- 0.000	0.00 +- 0.000	0.32 +- 0.016
16	11.35 +- 0.205	10.85 +- 0.187	0.01 +- 0.010	0.00 +- 0.000	0.49 +- 0.054

BASE-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	43.74 +- 0.000	28.95 +- 0.000	1.95 +- 0.000	12.33 +- 0.000	0.52 +- 0.000
2	33.10 +- 0.080	17.93 +- 0.269	3.78 +- 0.245	9.75 +- 0.436	1.65 +- 0.002
4	25.90 +- 0.043	12.23 +- 2.538	3.85 +- 0.988	7.69 +- 2.485	2.14 +- 0.953
8	34.34 +- 0.182	14.29 +- 4.556	11.23 +- 3.775	6.55 +- 5.836	2.27 +- 2.258
16	48.73 +- 0.498	18.79 +- 6.544	22.83 +- 5.537	5.15 +- 8.275	1.96 +- 3.274

BASE-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	43.71 +- 0.000	28.92 +- 0.000	1.95 +- 0.000	12.33 +- 0.000	0.52 +- 0.000
2	32.93 +- 0.069	19.29 +- 0.205	3.78 +- 0.387	8.35 +- 0.610	1.50 +- 0.087
4	24.14 +- 0.039	15.36 +- 0.284	2.86 +- 0.733	4.48 +- 0.208	1.44 +- 0.680
8	32.39 +- 0.239	18.88 +- 1.074	8.62 +- 2.821	3.02 +- 1.326	1.86 +- 2.062
16	41.21 +- 0.491	23.51 +- 1.254	14.52 +- 3.470	1.77 +- 1.579	1.41 +- 2.570

BASE-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	50.12 +- 0.000	31.59 +- 0.000	1.95 +- 0.000	16.06 +- 0.000	0.52 +- 0.000
2	41.50 +- 0.096	20.70 +- 0.150	5.64 +- 2.315	14.43 +- 2.371	0.73 +- 0.002
4	28.30 +- 0.075	18.89 +- 0.759	1.48 +- 0.174	7.45 +- 0.968	0.48 +- 0.002
8	34.39 +- 0.239	17.90 +- 5.397	8.62 +- 2.821	6.00 +- 5.812	1.86 +- 2.062
16	42.21 +- 0.491	22.20 +- 6.437	14.52 +- 3.470	4.08 +- 6.820	1.41 +- 2.570

BASE+WSHR-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	43.74 +- 0.000	28.95 +- 0.000	1.95 +- 0.000	12.33 +- 0.000	0.52 +- 0.000
2	33.10 +- 0.080	17.93 +- 0.269	3.78 +- 0.245	9.75 +- 0.436	1.65 +- 0.002
4	25.90 +- 0.043	12.23 +- 2.538	3.85 +- 0.988	7.69 +- 2.485	2.14 +- 0.953
8	34.34 +- 0.182	14.29 +- 4.556	11.23 +- 3.775	6.55 +- 5.836	2.27 +- 2.258
16	48.73 +- 0.498	18.79 +- 6.544	22.83 +- 5.537	5.15 +- 8.275	1.96 +- 3.274

BASE+WSHR-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	43.71 +- 0.000	28.92 +- 0.000	1.95 +- 0.000	12.33 +- 0.000	0.52 +- 0.000

Table B.8: (continued)

2	32.93 +- 0.069	19.29 +- 0.205	3.78 +- 0.387	8.35 +- 0.610	1.50 +- 0.087
4	24.14 +- 0.039	15.36 +- 0.284	2.86 +- 0.733	4.48 +- 0.208	1.44 +- 0.680
8	32.39 +- 0.239	18.88 +- 1.074	8.62 +- 2.821	3.02 +- 1.326	1.86 +- 2.062
16	41.21 +- 0.491	23.51 +- 1.254	14.52 +- 3.470	1.77 +- 1.579	1.41 +- 2.570

BASE+WSHR-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	50.12 +- 0.000	31.59 +- 0.000	1.95 +- 0.000	16.06 +- 0.000	0.52 +- 0.000
2	41.50 +- 0.096	20.70 +- 0.150	5.64 +- 2.315	14.43 +- 2.371	0.73 +- 0.002
4	28.30 +- 0.075	18.89 +- 0.759	1.48 +- 0.174	7.45 +- 0.968	0.48 +- 0.002
8	34.39 +- 0.239	17.90 +- 5.397	8.62 +- 2.821	6.00 +- 5.812	1.86 +- 2.062
16	42.21 +- 0.491	22.20 +- 6.437	14.52 +- 3.470	4.08 +- 6.820	1.41 +- 2.570

BASE+WSTL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	43.89 +- 0.000	29.10 +- 0.000	1.95 +- 0.000	12.33 +- 0.000	0.52 +- 0.000
2	27.00 +- 0.000	16.39 +- 0.883	1.91 +- 0.557	8.00 +- 0.344	0.69 +- 0.018
4	20.80 +- 0.011	12.95 +- 0.687	1.71 +- 0.449	5.29 +- 0.729	0.84 +- 0.385
8	23.09 +- 0.031	15.60 +- 2.061	3.00 +- 0.873	3.60 +- 2.058	0.88 +- 0.831
16	32.78 +- 0.200	19.03 +- 2.891	10.04 +- 2.472	2.81 +- 3.654	0.90 +- 1.466

BASE+WSTL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	44.20 +- 0.000	29.41 +- 0.000	1.95 +- 0.000	12.33 +- 0.000	0.52 +- 0.000
2	27.26 +- 0.002	17.45 +- 0.929	1.89 +- 0.573	7.31 +- 0.406	0.62 +- 0.052
4	20.63 +- 0.014	14.43 +- 0.122	1.46 +- 0.364	4.10 +- 0.122	0.64 +- 0.327
8	23.44 +- 0.048	17.88 +- 0.445	2.63 +- 0.788	2.27 +- 0.418	0.66 +- 0.677
16	30.71 +- 0.198	22.72 +- 0.508	6.06 +- 1.440	1.32 +- 0.605	0.62 +- 1.051

BASE+WSTL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	50.15 +- 0.000	31.63 +- 0.000	1.95 +- 0.000	16.06 +- 0.000	0.52 +- 0.000
2	35.76 +- 0.001	19.88 +- 0.500	4.07 +- 1.249	11.49 +- 1.769	0.33 +- 0.021
4	26.35 +- 0.006	19.00 +- 0.154	1.16 +- 0.159	6.00 +- 0.247	0.18 +- 0.015
8	33.75 +- 0.042	28.84 +- 0.868	1.19 +- 0.238	3.58 +- 0.991	0.14 +- 0.030
16	40.26 +- 0.162	36.30 +- 1.361	1.66 +- 0.438	2.17 +- 1.456	0.12 +- 0.039

BASE+WSTLGBL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	43.85 +- 0.000	29.00 +- 0.000	1.95 +- 0.000	12.33 +- 0.000	0.58 +- 0.000
2	31.33 +- 0.004	18.78 +- 0.459	2.77 +- 1.367	7.97 +- 0.694	1.80 +- 0.218
4	26.70 +- 0.078	17.21 +- 0.753	2.53 +- 0.592	5.17 +- 0.782	1.78 +- 0.737
8	25.28 +- 0.148	17.03 +- 1.774	3.46 +- 0.852	3.45 +- 1.805	1.34 +- 1.014

Table B.8: (continued)

16	35.00 +- 0.215	22.48 +- 3.110	8.59 +- 1.910	2.79 +- 3.473	1.14 +- 1.505

BASE+WSTLGBL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	43.87 +- 0.000	29.02 +- 0.000	1.95 +- 0.000	12.33 +- 0.000	0.58 +- 0.000
2	31.40 +- 0.006	19.82 +- 0.463	2.68 +- 1.305	7.21 +- 0.584	1.68 +- 0.252
4	26.32 +- 0.068	18.10 +- 0.542	2.38 +- 0.551	4.17 +- 0.236	1.67 +- 0.721
8	26.04 +- 0.139	19.04 +- 0.818	3.33 +- 0.783	2.36 +- 0.448	1.31 +- 1.062
16	32.99 +- 0.260	25.21 +- 0.896	5.58 +- 1.160	1.32 +- 0.600	0.87 +- 1.193

BASE+WSTLGBL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	50.48 +- 0.000	31.89 +- 0.000	1.95 +- 0.000	16.06 +- 0.000	0.58 +- 0.000
2	38.65 +- 0.006	23.11 +- 0.086	3.55 +- 0.230	10.55 +- 0.068	1.44 +- 0.218
4	37.01 +- 0.068	26.92 +- 0.911	2.79 +- 0.834	6.09 +- 0.485	1.21 +- 0.772
8	32.92 +- 0.171	26.45 +- 0.664	2.18 +- 0.627	3.59 +- 0.986	0.69 +- 0.819
16	40.46 +- 0.261	35.52 +- 1.213	2.43 +- 0.538	2.14 +- 1.321	0.37 +- 0.689

BASE+VC-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	42.21 +- 0.000	34.21 +- 0.000	1.95 +- 0.000	5.54 +- 0.000	0.52 +- 0.000
2	30.94 +- 0.058	20.68 +- 0.044	3.66 +- 0.071	5.01 +- 0.081	1.59 +- 0.004
4	19.94 +- 0.067	13.67 +- 0.622	2.03 +- 0.879	3.08 +- 1.042	1.16 +- 0.390
8	30.34 +- 0.256	18.34 +- 0.948	8.09 +- 2.986	2.82 +- 2.413	1.09 +- 0.936
16	28.87 +- 0.238	21.47 +- 2.135	5.14 +- 1.306	1.63 +- 2.269	0.63 +- 0.885

BASE+VC-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	42.22 +- 0.000	34.22 +- 0.000	1.95 +- 0.000	5.54 +- 0.000	0.52 +- 0.000
2	25.33 +- 0.023	20.05 +- 0.026	1.41 +- 0.307	3.14 +- 0.311	0.73 +- 0.003
4	17.36 +- 0.007	14.26 +- 0.377	0.93 +- 0.225	1.69 +- 0.156	0.48 +- 0.002
8	18.61 +- 0.055	16.31 +- 0.581	1.05 +- 0.556	0.98 +- 0.082	0.27 +- 0.002
16	21.30 +- 0.065	19.19 +- 0.394	1.41 +- 0.366	0.57 +- 0.046	0.14 +- 0.003

BASE+VC-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	42.26 +- 0.000	34.26 +- 0.000	1.95 +- 0.000	5.54 +- 0.000	0.52 +- 0.000
2	25.48 +- 0.034	19.95 +- 0.372	1.36 +- 0.261	3.45 +- 0.601	0.73 +- 0.002
4	17.19 +- 0.017	13.97 +- 0.426	0.71 +- 0.140	2.03 +- 0.327	0.48 +- 0.002
8	18.26 +- 0.021	15.91 +- 0.667	0.86 +- 0.597	1.21 +- 0.175	0.27 +- 0.002
16	21.52 +- 0.077	19.19 +- 0.441	1.39 +- 0.370	0.80 +- 0.104	0.14 +- 0.002

Table B.8: (continued)

BASE+VC+WSHR-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	42.21 +- 0.000	34.21 +- 0.000	1.95 +- 0.000	5.54 +- 0.000	0.52 +- 0.000
2	30.94 +- 0.058	20.68 +- 0.044	3.66 +- 0.071	5.01 +- 0.081	1.59 +- 0.004
4	19.94 +- 0.067	13.67 +- 0.622	2.03 +- 0.879	3.08 +- 1.042	1.16 +- 0.390
8	30.34 +- 0.256	18.34 +- 0.948	8.09 +- 2.986	2.82 +- 2.413	1.09 +- 0.936
16	28.87 +- 0.238	21.47 +- 2.135	5.14 +- 1.306	1.63 +- 2.269	0.63 +- 0.885

BASE+VC+WSHR-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	42.22 +- 0.000	34.22 +- 0.000	1.95 +- 0.000	5.54 +- 0.000	0.52 +- 0.000
2	25.33 +- 0.023	20.05 +- 0.026	1.41 +- 0.307	3.14 +- 0.311	0.73 +- 0.003
4	17.36 +- 0.007	14.26 +- 0.377	0.93 +- 0.225	1.69 +- 0.156	0.48 +- 0.002
8	18.61 +- 0.055	16.31 +- 0.581	1.05 +- 0.556	0.98 +- 0.082	0.27 +- 0.002
16	21.30 +- 0.065	19.19 +- 0.394	1.41 +- 0.366	0.57 +- 0.046	0.14 +- 0.003

BASE+VC+WSHR-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	42.26 +- 0.000	34.26 +- 0.000	1.95 +- 0.000	5.54 +- 0.000	0.52 +- 0.000
2	25.48 +- 0.034	19.95 +- 0.372	1.36 +- 0.261	3.45 +- 0.601	0.73 +- 0.002
4	17.19 +- 0.017	13.97 +- 0.426	0.71 +- 0.140	2.03 +- 0.327	0.48 +- 0.002
8	18.26 +- 0.021	15.91 +- 0.667	0.86 +- 0.597	1.21 +- 0.175	0.27 +- 0.002
16	21.52 +- 0.077	19.19 +- 0.441	1.39 +- 0.370	0.80 +- 0.104	0.14 +- 0.002

BASE+VC+WSTL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	42.23 +- 0.000	34.23 +- 0.000	1.95 +- 0.000	5.54 +- 0.000	0.52 +- 0.000
2	23.83 +- 0.007	18.88 +- 0.239	1.45 +- 0.414	3.06 +- 0.154	0.44 +- 0.014
4	16.79 +- 0.015	13.38 +- 0.482	1.07 +- 0.347	1.93 +- 0.343	0.40 +- 0.149
8	17.55 +- 0.033	15.39 +- 0.437	0.76 +- 0.175	1.10 +- 0.419	0.30 +- 0.173
16	24.65 +- 0.065	21.36 +- 1.052	1.98 +- 0.484	0.97 +- 1.000	0.33 +- 0.398

BASE+VC+WSTL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	42.30 +- 0.000	34.30 +- 0.000	1.95 +- 0.000	5.54 +- 0.000	0.52 +- 0.000
2	23.52 +- 0.006	19.28 +- 0.234	1.09 +- 0.120	2.83 +- 0.103	0.32 +- 0.004
4	16.23 +- 0.015	13.93 +- 0.129	0.62 +- 0.031	1.49 +- 0.145	0.19 +- 0.011
8	16.08 +- 0.012	14.74 +- 0.122	0.42 +- 0.040	0.79 +- 0.088	0.14 +- 0.031
16	18.75 +- 0.051	17.73 +- 0.126	0.48 +- 0.088	0.44 +- 0.033	0.10 +- 0.025

BASE+VC+WSTL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	42.52 +- 0.000	34.39 +- 0.000	1.95 +- 0.000	5.54 +- 0.000	0.64 +- 0.000

Table B.8: (continued)

2	23.14 +- 0.020	18.96 +- 0.079	0.98 +- 0.003	2.88 +- 0.104	0.32 +- 0.002
4	16.34 +- 0.006	14.04 +- 0.207	0.53 +- 0.032	1.60 +- 0.170	0.17 +- 0.010
8	16.34 +- 0.011	15.15 +- 0.145	0.28 +- 0.035	0.83 +- 0.102	0.09 +- 0.006
16	19.17 +- 0.061	18.29 +- 0.139	0.30 +- 0.078	0.54 +- 0.060	0.05 +- 0.002

BASE+VC+WSTLGBL-MSG

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	42.31 +- 0.000	34.25 +- 0.000	1.95 +- 0.000	5.54 +- 0.000	0.58 +- 0.000
2	29.69 +- 0.001	22.30 +- 0.768	2.63 +- 1.511	3.17 +- 0.527	1.58 +- 0.214
4	22.83 +- 0.046	17.39 +- 0.303	1.93 +- 0.583	2.09 +- 0.356	1.41 +- 0.702
8	21.30 +- 0.123	16.97 +- 1.092	1.74 +- 0.439	1.57 +- 0.937	1.02 +- 0.839
16	31.07 +- 0.179	24.83 +- 1.887	3.98 +- 1.067	1.46 +- 1.938	0.79 +- 0.976

BASE+VC+WSTLGBL-PUT/GET

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	42.28 +- 0.000	34.22 +- 0.000	1.95 +- 0.000	5.54 +- 0.000	0.58 +- 0.000
2	28.84 +- 0.001	22.35 +- 0.442	2.21 +- 1.076	2.85 +- 0.420	1.42 +- 0.215
4	21.70 +- 0.048	17.45 +- 0.334	1.55 +- 0.589	1.54 +- 0.126	1.15 +- 0.747
8	19.80 +- 0.105	17.05 +- 0.627	1.12 +- 0.320	0.94 +- 0.089	0.68 +- 0.813
16	22.21 +- 0.145	20.14 +- 0.570	1.13 +- 0.260	0.56 +- 0.071	0.37 +- 0.691

BASE+VC+WSTLGBL-LD/ST

N	TOTAL	COMPUTE	IDLE	COHERENCE	OVERHEAD
1	42.52 +- 0.000	34.34 +- 0.000	1.95 +- 0.000	5.54 +- 0.000	0.70 +- 0.000
2	26.88 +- 0.907	21.34 +- 0.507	1.47 +- 0.469	3.01 +- 0.203	1.06 +- 0.271
4	19.62 +- 0.148	16.00 +- 0.503	0.94 +- 0.296	1.58 +- 0.142	1.11 +- 0.841
8	23.77 +- 0.272	19.37 +- 1.379	2.07 +- 0.913	0.98 +- 0.118	1.35 +- 1.719
16	33.08 +- 0.382	28.21 +- 2.724	2.90 +- 2.273	0.59 +- 0.069	1.37 +- 2.126

REFERENCES

[1] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of Computational Physics*, vol. 53, pp. 484–512, 1984.

[2] M. J. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics," *Journal of Computational Physics*, vol. 82, pp. 64–84, 1989.

[3] E. Steinthorsson, D. Modiano, W. Y. Crutchfield, J. B. Bell, and P. Colella, "Three dimensional adapative mesh refinement for hyperbolic conservation law," in *Proceedings of the 12th AIAA Computational Fluid Dynamics Conference*, 1995, pp. 902–912.

[4] J. Barnes and P. Hut, "A hierarchical O(N log N) force calculation algorithm," The Institute for Advanced Study, Princeton, New Jersey, tech. rep., 1986.

[5] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of Computational Physics*, vol. 73, pp. 325–48, 1987.

[6] M. Haines, D. Cronk, and P. Mehrotra, "On the design of Chant: A talking threads package," in *Proceedings of Supercomputing'94*, 1994, pp. 350–359.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, A. Shaw, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of Principles and Practice of Parallel Programming*, 1995, pp. 207–216.

[8] D. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek, "Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages an Operating Systems*, 1991, pp. 164–75.

[9] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren, "Supporting dynamic data structures on distributed memory machines," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 2, pp. 233–263, 1995.

[10] R. Chandra, A. Gupta, and J. L. Hennessy, "Data locality and load balancing in COOL," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993, pp. 249–259.

[11] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews, "Distributed filaments: Efficient fine-grain parallelism on a cluster of workstations," in *First Symposium on Operating Systems Design and Implementation*, 1994, pp. 201–212.

[12] L. V. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proceedings of OOPSLA'93*, 1993, pp. 91–108.

[13] I. Foster, C. Kesselman, R. Olson, and S. Tuecke, “Nexus: An interoperability layer for parallel and distributed computer systems,” Argonne National Laboratory, Tech. Rep. Version 1.3, 1993.

[14] A. Grimshaw, “Easy-to-use object-oriented parallel processing with Mentat,” *IEEE Computer*, vol. 5, no. 26, pp. 39–51, 1993.

[15] A. A. Chien, *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs*. Cambridge, MA: MIT Press, 1993.

[16] A. Yonezawa, ed., *ABCL: An Object-Oriented Concurrent System*. Cambridge, MA: MIT Press, 1990.

[17] A. Chien and U. Reddy, “ICC++ language definition.” Concurrent Systems Architecture Group Memo, Also available from http://www-csag.cs.uiuc.edu, 1995.

[18] J. Plevyak, “Optimization of object-oriented and concurrent programs,” Ph.D. dissertation, University of Illinois, Urbana, IL. 1996.

[19] A. Chien, V. Karamcheti, and J. Plevyak, “The Concert system—compiler and runtime support for efficient fine-grained concurrent object-oriented programs,” Department of Computer Science, University of Illinois, Urbana, IL, Tech. Rep. UIUCDCS-R-93-1815, 1993.

[20] A. Chien, V. Karamcheti, J. Plevyak, and D. Sahrawat, “The Concert system: Compiler and runtime technology for efficient concurrent object-oriented programming,” in *Proceedings of the Computing in Aerospace 9 Conference*, 1993, pp. 709–720.

[21] A. Chien, J. Dolby, B. Ganguly, V. Karamcheti, and X. Zhang, “Evaluating high level parallel programming support for irregular applications in ICC++,” in *Proceedings of the International Scientific Computing in Object-oriented Parallel Environments Conference*, 1997, pp. 121–128.

[22] A. Chien, J. Dolby, B. Ganguly, V. Karamcheti, and X. Zhang, “Supporting high level programming with high performance: The Illinois Concert system,” in *Proceedings of the Second International Workshop on High-level Parallel Programming Models and Supportive Environments*, 1997, pp. 15–24.

[23] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, “Active Messages: a mechanism for integrated communication and computation,” in *Proceedings of the International Symposium on Computer Architecture*, 1992, pp. 256–266.

[24] E. Mohr, D. Kranz, and R. Halstead Jr., “Lazy task creation: A technique for increasing the granularity of parallel programs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 264–280, 1991.

[25] D. B. Wagner and B. G. Calder, “Leapfrogging: A portable technique for implementing efficient futures,” in *Proceedings of the Fifth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1993, pp. 208–217.

[26] S. C. Goldstein, K. E. Schauser, and D. Culler, “Lazy threads: Implementing a fast parallel call,” *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 5–20, 1996.

[27] K. Taura, S. Matsuoka, and A. Yonezawa, "StackThreads: An abstract machine for scheduling fine-grain threads on stock CPUs," in *Proceedings of Workshop on Theory and Practice of Parallel Programming*, 1994, pp. 121–136.

[28] J. Carter, J. Bennett, and W. Zwaenepoel, "Implementation and performance of Munin," in *Proceedings of the ACM Symposium on Operating Systems Principles*, 1991, pp. 152–164.

[29] B. K. Totty, "Tunable shared memory abstractions for distributed memory systems," Ph.D. dissertation, University of Illinois, Urbana-Champaign, IL, 1994.

[30] M. D. Hill, J. R. Larus, and D. A. Wood, "Tempest: A substrate for portable parallel programs," in *Compcon*, 1995, pp. 321–324.

[31] R. H. Halstead Jr., "Multilisp: A language for concurrent symbolic computation," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 4, pp. 501–538, 1985.

[32] A. A. Chien, U. S. Reddy, J. Plevyak, and J. Dolby, "ICC++ – a C++ dialect for high-performance parallel computation," in *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software*, 1996, pp. 76–95.

[33] S. Murer, J. A. Feldman, C.-C. Lim, and M.-M. Seidel, "pSather: Layered extensions to an object-oriented language for efficient parallel computation," International Computer Science Institute, Berkeley, CA, Tech. Rep. TR-93-028, 1993.

[34] C. A. R. Hoare, "Communicating sequential processes," *Communications of the Association for Computing Machinery*, vol. 21, no. 8, pp. 666–677, 1978.

[35] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: MIT Press, 1986.

[36] B. J. Smith, "A pipelined, shared resource MIMD computer," in *IEEE Proceeding of the International Conference on Parallel Processing*, 1978, pp. 6–8.

[37] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera computer system," in *International Conference on Supercomputing*, 1990, pp. 1–6.

[38] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *22nd Annual International Symposium on Computer Architecture*, 1995, pp. 392–403.

[39] *The Connection Machine CM-5 Technical Summary*, 1991, Available from Thinking Machines Corporation, Cambridge, MA.

[40] *Cray T3D System Architecture Overview*, 1993, Available from Cray Research, Inc., Eagan, MN.

[41] *Origin Servers: Technical Overview of the Origin Family*, 1996, Available from Silicon Graphics, Inc., Mountain View, CA.

[42] S. L. Scott, "Synchronization and communication in the T3E multiprocessor," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996, pp. 26–36.

[43] *Convex Exemplar Architecture*, 1993, Available from Convex Computer Corporation, Richardson, TX.

[44] *Scalable POWERparallel System*, 1995, Available from IBM Corporation, White Plains, NY.

[45] B. Stroustrup, *The C++ Programming Language*, 2nd ed. Reading, MA: Addison Wesley, 1991.

[46] A. Chien, J. Dolby, V. Karamcheti, J. Plevyak, and X. Zhang, "The ICC++ reference manual." Concurrent Systems Architecture Group Memo. Available from http://www-csag.cs.uiuc.edu/, 1996.

[47] P. Hanrahan, D. Salzman, and L. Aupperle, "A rapid hierarchical radiosity algorithm," *Computer Graphics (Proc Siggraph)*, vol. 25, no. 4, pp. 197–206, 1991.

[48] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. G. Steinberg, and K. Yelick, "Empirical evaluation of the CRAY-T3D: A compiler perspective," in *Proceeedings of the International Symposium on Computer Architecture*, 1995, pp. 320–331.

[49] R. Numrich, P. Springer, and J. Peterson, "Measurement of communication rates on the cray T3D interconnection network," in *Proceedings of HPCN Europe'94, Vol. II*, 1994, pp. 150–157.

[50] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA highly scalable server," *Computer Architecture News*, vol. 25, no. 2, pp. 241–251, 1997.

[51] S. Scott and G. Thorson, "Optimized routing in the Cray T3D," in *Proceedings of the Parallel Computer Routing and Communication Workshop (PCRCW'94)*, 1994, pp. 281–294.

[52] *DECchip 21064-AA Microprocessor Hardware Reference Manual*, 1992, Available from Digital Equipment Corporation, Maynard, MA.

[53] R. W. Numrich, "The Cray T3D address space and how to use it," Cray Research, Inc., Tech. Rep., 1994.

[54] *CRAY T3D Software Overview Technical Note*, 1992, Available from Cray Research, Inc., Eagan, MN.

[55] M. Galles, "Spider: A high-speed network interconnect," *IEEE Micro*, vol. 17, no. 1, pp. 34–39, 1997.

[56] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–40, 1996.

[57] J. Laudon and D. Lenoski, "System overview of the SGI Origin 200/2000 product line," in *Proceedings of COMPCON 97*, 1997, pp. 150–156.

[58] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the dash multiprocessor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 148–159.

[59] W.-D. Weber, "Scalable directories for cache-coherent shared-memory multiprocessors," Ph.D. dissertation, Stanford University, Stanford, CA, 1993.

[60] J. Dolby, “ParaSight: A debugger for concurrent object-oriented programs,” M.S. thesis, University of Illinois, Urbana, IL, 1995.

[61] A. A. Chien and J. Dolby, “The Illinois Concert system: A problem-solving environment for irregular applications,” in *Proceedings of DAGS'94, The Symposium on Parallel Computation and Problem Solving Environments.*, 1994, pp. 21–29.

[62] A. A. Chien, W. Feng, V. Karamcheti, and J. Plevyak, “Techniques for efficient execution of fine-grained concurrent programs,” in *Proceedings of the Fifth Workshop on Compilers and Languages for Parallel Computing*, 1992, pp. 103–113.

[63] J. Plevyak, V. Karamcheti, and A. Chien, “Analysis of dynamic structures for efficient parallel execution,” in *Proceedings of the Sixth Workshop for Languages and Compilers for Parallel Machines*, 1993, pp. 37–56.

[64] J. Plevyak and A. Chien, “Incremental inference of concrete types,” Department of Computer Science, University of Illinois, Urbana, IL, Tech. Rep. UIUCDCS-R-93-1829, 1993.

[65] J. Plevyak and A. A. Chien, “Precise concrete type inference of object-oriented programs,” in *Proceedings of OOPSLA'94, Object-Oriented Programming Systems, Languages and Architectures*, 1994, pp. 324–340.

[66] J. Plevyak, X. Zhang, and A. A. Chien, “Obtaining sequential efficiency in concurrent object-oriented programs,” in *Proceedings of the ACM Symposium on the Principles of Programming Languages*, 1995, pp. 311–321.

[67] J. Plevyak and A. A. Chien, “Type directed cloning for object-oriented programs,” in *Proceedings of the Workshop for Languages and Compilers for Parallel Computing*, 1995, pp. 566–580.

[68] J. Dolby, “Automatic inline allocation of objects,” in *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997, pp. 7–17.

[69] X. Zhang and A. A. Chien, “Dynamic pointer alignment: Tiling and communication optimizations for parallel pointer-based computations,” in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1997, pp. 37–47.

[70] V. Karamcheti, J. Plevyak, and A. A. Chien, “Runtime mechanisms for efficient dynamic multithreading,” *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 21–40, 1996.

[71] V. Karamcheti and A. A. Chien, “View caching: Efficient software shared memory for dynamic computations,” in *Proceedings of the International Parallel Processing Symposium*, 1997, pp. 483–489.

[72] X. Zhang, V. Karamcheti, T. Ng, and A. Chien, “Optimizing COOP languages: Study of a protein dynamics program,” in *IPPS'96*, 1996, pp. 235–240.

[73] A. A. Chien, M. Straka, J. Dolby, V. Karamcheti, J. Plevyak, and X. Zhang, “A case study in irregular parallel programming,” in *Proceedings of the DIMACS Workshop on Specification of Parallel Algorithms*, 1994, pp. 1–20.

[74] S. Pakin, V. Karamcheti, and A. A. Chien, “Fast Messages: Efficient, portable communication for workstation clusters and MPPs,” *IEEE Concurrency*, vol. 5, no. 2, pp. 60–73, 1997.

[75] A. B. Sinha, L. V. Kale, and B. Ramkumar, “A dynamic and adaptive quiescence detection algorithm,” Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana, IL, Tech. Rep. 93-11, 1993.

[76] H. Baker, “The Treadmill: Real-time garbage collection without motion sickness,” *ACM SIGPLAN Notices*, vol. 27, no. 3, pp. 66–70, 1992.

[77] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, “The Midway distributed shared memory system,” in *Proceedings of COMPCON 1993*, 1993, pp. 528–537.

[78] D. J. Scales and M. S. Lam, “The design and evaluation of a shared object system for distributed memory machines,” in *First Symposium on Operating Systems Design and Implementation*, 1994, pp. 101–114.

[79] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach, “CRL: High-performance all-software distributed shared memory,” in *Proceedings of the Symposium on Operating Systems Principles*, 1995, pp. 213–228.

[80] Message Passing Interface Forum, “MPI: A message passing interface standard,” University of Tennessee, Knoxville, Tech. Rep. Version 1.1, 1995.

[81] “Thread extensions for portable operating systems.” IEEE POSIX 1003.1.c Standard, 1995.

[82] I. Foster, C. Kesselman, and S. Tuecke, “The Nexus approach to integrating multithreading and communication,” *J. Parallel and Distributed Computing*, vol. 37, no. 1, pp. 70–82, 1996.

[83] S. Pakin, M. Lauria, and A. Chien, “High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet,” in *Supercomputing '95*, 1995, pp. 501–511.

[84] V. Karamcheti and A. A. Chien, “A comparison of architectural support for messaging on the TMC CM-5 and the Cray T3D,” in *Proceedings of the International Symposium on Computer Architecture*, 1995, pp. 298–307.

[85] V. Karamcheti and A. Chien, “Concert—efficient runtime support for concurrent object-oriented programming languages on stock hardware,” in *Proceedings of Supercomputing '93*, 1993, pp. 598–607.

[86] L. V. Kale, B. Ramkumar, V. Saletore, and A. B. Sinha, “Prioritization in parallel symbolic computing,” *Lecture Notes in Computer Science*, vol. 748, pp. 12–41, 1993.

[87] *PVM 3 User's Guide and Reference Manual*, 1994, A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. Oak Ridge National Laboratory, Oak Ridge, TN 37831.

[88] L. V. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. M. Yelon, “Converse: an interoperable framework for parallel programming,” in *Proceedings of the International Parallel Processing Symposium*, 1996, pp. 212–217.

[89] PORTS Consortium, “The PORTS0 interface,” Argonne National Laboratory, Tech. Rep. Version 0.3, 1995.

[90] B. Bershad, E. Lazowska, and H. Levy, “Presto: A system for object-oriented parallel programming,” *Software—Practice and Experience*, vol. 18, no. 8, pp. 713–732, 1988.

[91] A. Chatterjee, A. Khanna, and Y. Hung, "ES-Kit: An object-oriented distributed system," *Concurrency - Practice and Experience*, vol. 3, no. 6, pp. 525–540, 1991.

[92] K. M. Chandy and C. Kesselman, "Compositional C++: Compositional parallel programming," in *Proceedings of the Fifth Workshop on Compilers and Languages for Parallel Computing*, 1992, pp. 114–123.

[93] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda, "Object-oriented concurrent programming – modelling and programming in an object-oriented concurrent language ABCL/1," in *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, Eds., pp. 55–89, Cambridge, MA: MIT Press, 1987.

[94] C. Houck and G. Agha, "HAL: A high-level actor language and its distributed implementation," in *Proceedings of the 21st International Conference on Parallel Processing*, 1992, pp. 158–165.

[95] L. V. Kale, "The Chare-Kernel parallel programming language," in *Proceedings of the International Conference on Parallel Processing*, 1990, pp. 17–25.

[96] W. C. Hsieh, P. Wang, and W. E. Weihl, "Computation migration: Enhancing locality for distributed-memory parallel systems," in *Proceedings of the Fifth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1993, pp. 239–248.

[97] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, "Run-time scheduling and execution of loops on message passing machines," *Journal of Parallel and Distributed Computing*, vol. 8, pp. 303–312, 1990.

[98] S. Haranandani, J. Saltz, P. Mehrotra, and H. Berryman, "Performance of hashed cache data migration schemes on multicomputers," *Journal of Parallel and Distributed Computing*, vol. 12, pp. 415–422, 1991.

[99] S. D. Sharma, R. Ponnusamy, B. Moon, Y. shin Hwang, R. Das, and J. Saltz, "Run-time and compile-time support for adaptive irregular problems," in *Proceedings of Supercomputing '94*, 1994, pp. 97–106.

[100] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems*, vol. 7, no. 4, pp. 321–359, 1989.

[101] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenopol, "Treadmarks: Distributed shared memory on standard workstations and operating systems," in *Proceedings of the 1994 Winter Usenix Conference*, 1994, pp. 115–132.

[102] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li, "Improving release-consistent shared virtual memory using automatic update," in *Proceedings of the 2nd International Symposium on High Performance Computer Architecture*, 1996, pp. 14–25.

[103] R. Raj, E. Tempero, H. Levy, A. Black, N. Hutchinson, and E. Jul, "Emerald: A general-purpose programming language," *Software Practice and Experience*, vol. 21, no. 1, pp. 91–118, 1991.

[104] J. S. Chase, F. G. Amador, E. Lazowska, H. Levy, and R. J. Littlefield, "The Amber system: Parallel programming on a network of multiprocessors," in *Proceedings of Twelfth Symposium on Operating Systems Principles*, 1989, pp. 147–58.

[105] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "Orca: A language for parallel programming of distributed systems," *IEEE Transactions on Software Engineering*, vol. 18, no. 3, pp. 190–205, 1992.

[106] M. J. Feeley and H. M. Levy, "Distributed shared memory with versioned objects," in *Proceedings of OOPSLA*, 1992, pp. 247–262.

[107] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. Wood, "Fine-grain access control for distributed shared memory," in *ASPLOS-VI*, 1994, pp. 297–306.

[108] D. Scales, K. Gharachorloo, and C. Thekkath, "Shasta: A low overhead, software-only approach for supporting fine-grain shared memory," in *Proceedings of ASPLOS-VII*, 1996, pp. 174–185.

[109] K. Gharacharloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *IEEE Computer*, pp. 15–26, 1990.

[110] J. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Proceedings of the Second ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1990, pp. 168–176.

[111] S. K. Reinhardt, "Tempest interface specification." Revision 1.2.1, Available from http://www.cs.wisc.edu, 1995.

[112] K. Petersen and K. Li, "Multiprocessor cache coherence based on virtual memory support," *Journal of Parallel and Distributed Computing*, vol. 29, no. 2, pp. 158–178, 1995.

[113] L. I. Kontothanassis and M. L. Scott, "Using memory-mapped network interfaces to improve the performance of distributed shared memory," in *Proceedings of the 2nd International Symposium on High Performance Computer Architecture*, 1996, pp. 166–177.

[114] J. A. Stankovic, "Simulations of three adaptive, decentralized controlled, job scheduling algorithms," *Computer Networks*, vol. 8, no. 3, pp. 199–217, 1984.

[115] Y. Wang and R. J. T. Morris, "Load sharing in distributed systems," *IEEE Transactions on Computers*, vol. C-34, no. 3, pp. 204–217, 1985.

[116] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 5, pp. 662–674, 1986.

[117] C.-P. Wen, S. Chakrabarti, E. Deprit, A. Krishnamurthy, and K. Yelick, "Run-time support for portable distributed data structures," in *Proceedins of Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, 1995, pp. 111–120.

[118] R. Keller and F. C. H. Lin, "Simulated performance of a reduction based multiprocessor," *IEEE Computer*, vol. 17, no. 7, 1984.

[119] L. V. Kale, "Comparing the performance of two dynamic load distribution methods," in *International Conference on Parallel Processing*, 1988, pp. 8–12.

[120] W. Shu and L. V. Kale, "A dynamic scheduling strategy for the Chare Kernel system," in *Proceedings of Supercomputing'89*, 1989, pp. 389–398.

[121] R. M. Karp and Y. Zhang, “Randomized parallel algorithms for backtrack search and branch-and-bound computation,” *Journal of the Association of Computing Machinery*, vol. 40, no. 3, pp. 765–789, 1993.

[122] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, 1994, pp. 356–368.

[123] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg, “Virtual memory mapped network interface for the SHRIMP multicomputer,” in *Proceeding of the International Symposium on Computer Architecture*, 1994, pp. 142–153.

[124] R. B. Gillett, “Memory Channel network for PCI,” *IEEE Micro*, vol. 16, no. 1, pp. 12–18, 1996.

[125] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes, “An implementation of the Hamlyn sender-managed interface architecture,” in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, 1996, pp. 245–259.

[126] *IBM POWERparallel System - SP2, Performance Measurements*, 1995, J. M. Kuzela. Power Parallel Division, IBM.

[127] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, “Myrinet—a gigabit-per-second local-area network,” *IEEE Micro*, vol. 15, no. 1, pp. 29–36, 1995.

[128] D. Lenoski et al., “The Stanford DASH Multiprocessor,” *IEEE Computer*, pp. 63–79, 1992.

[129] J. Kuskin et al., “The Stanford FLASH Multiprocessor,” in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994, pp. 302–313.

[130] J. Kubiatowicz and A. Agarwal, “The anatomy of a message in the Alewife multiprocessor,” in *Proceedings of the International Conference on Supercomputing*, 1993, pp. 195–206.

[131] J. Plevyak, V. Karamcheti, X. Zhang, and A. Chien, “A hybrid execution model for fine-grained languages on distributed memory multicomputers,” in *Proceedings of Supercomputing'95*, 1995.

[132] W. Horwat, A. Chien, and W. Dally, “Experience with CST: Programming and implementation,” in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 1989, pp. 101–109.

[133] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin, “Sparcle: An evolutionary processor design for large-scale multiprocessors,” *IEEE Micro*, vol. 13, no. 3, pp. 48–61, 1993.

[134] W. J. Dally, A. Chien, S. Fiske, W. Horwat, J. Keen, M. Larivee, R. Lethin, P. Nuth, S. Wills, P. Carrick, and G. Fyler, “The J-Machine: A fine-grain concurrent computer,” in *Information Processing 89, Proceedings of the IFIP Congress*, 1989, pp. 1147–1153.

[135] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba, “An architecture of a dataflow single chip processor,” in *International Symposium on Computer Architecture*, 1989, pp. 46–53.

[136] R. S. Nikhil, G. M. Papadopoulos, and Arvind, "*T: A multithreaded massively parallel architecture," in *International Symposium on Computer Architecture*, 1992, pp. 156–167.

[137] K. Taura and A. Yonezawa, "Fine-grain multithreading with minimal compiler support — a cost effective approach to implementing efficient multithreading languages," in *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997, pp. 320–333.

[138] M. A. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina, "Virtual-memory-mapped network interfaces," *IEEE Micro*, pp. 21–28, 1995.

[139] G. Buzzard, D. Jacobson, S. Marovich, and J. Wilkes, "Hamlyn: A high-performance network interface with sender-based memory management," in *Proceedings of the IEEE Hot Interconnects Symposium*, 1995.

[140] G. F. Pfister and V. A. Norton, "Hot spot contention and combining in multistage interconnection networks," *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 943–948, 1985.

[141] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb, "Supporting systolic and memory communication in iWarp," in *Proceedings of the 17th International Symposium on Computer Architecture*, 1990, pp. 70–81.

[142] P. Druschel and L. L. Peterson, "Fbufs: A high-bandwidth cross-domain transfer facility," in *Proceedings of Fourteenth ACM Symposium on Operating Systems Principles*, 1993, pp. 189–202.

[143] H. L. C. A. Thekkath and E. D. Lazowska, "Separating data and control transfer in distributed operating systems," in *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994, pp. 2–11.

[144] A. Gottlieb et al., "The NYU Ultracomputer—designing an mimd shared memory parallel computer," *IEEE Transactions on Computers*, vol. C-32, no. 2, pp. 175–189, 1983.

[145] T. Callahan and S. Goldstein, "NIFDY: A low overhead, high throughput network interface," in *Proceedings of the International Symposium on Computer Architecture*, 1995, pp. 230–241.

[146] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. V. de Geijn, "Basic linear algebra communication subprograms," in *Sixth Distributed Memory Computing Conference Proceedings*, 1991, pp. 287–290.

[147] E. A. Brewer and B. C. Kuszmaul, "How to get good performance from the CM-5 data network," in *Proceedings of the International Parallel Processing Symposium*, 1994, pp. 858–867.

[148] A. L. Cox and R. J. Fowler, "Adaptive cache coherency for detecting migratory shared data," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993, pp. 109–118.

[149] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shareed-memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 12, pp. 87–106, 1991.

[150] H. S. Sandhu, B. Gamsa, and S. Zhou, "The shared region approach to software cache coherence on multiprocessors," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'93)*, 1993, pp. 229–238.

[151] K. L. Johnson, "High-performance all-software distributed shared memory," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, 1995.

[152] H. Cheong and A. V. Veidenbaum, "Compiler-directed cache management in multiprocessors," *IEEE Computer*, vol. 23, no. 6, pp. 39–48, 1990.

[153] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: Freeman, 1979.

[154] James Philbin et. al., "Thread scheduling for cache locality," in *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996, pp. 60–71.

[155] P. Goyal, X. Guo, and H. M. Vin, "A hierarchical CPU scheduler for multimedia operating systems," in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, 1996, pp. 107–121.

[156] C. A. Waldspurger, "Lottery and stride scheduling: Flexible proportional-share resource management," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, 1995.

[157] C. Waldspurger and W. Weihl, "Lottery scheduling: Flexible, proportional share resource management," in *Proceedings of the Symposium on Operating Systems Design and Implementation*, 1994, pp. 232–245.

[158] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the Association for Computing Machinery*, vol. 20, no. 1, pp. 46–61, 1973.

[159] J. Hyman, A. Lazar, and G. Pacifici, "Real-time scheduling with quality of service constraints," *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 7, pp. 1052–1063, 1991.

[160] I. Stoica, H. Abdel-Wahab, and K. Jeffay, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *Proceedings of IEEE Real Time Systems Symposium*, 1996, pp. 288–299.

[161] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, "Dag-consistent distributed shared memory," in *10th International Parallel Processing Symposium*, 1996, pp. 132–141.

[162] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiling FORTRAN D for MIMD distributed-memory machines," *Communications of the ACM*, vol. 35, no. 8, pp. 66–80, 1992.

[163] G. C. Fox, *Numerical Algorithms for Modern Parallel Computer Architectures*. New York: Springer-Verlag, 1988.

[164] H. Samet, *The Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley, 1990.

[165] J. P. Singh, “Parallel hierarchical N-body methods and their implications for multiprocessors,” Ph.D. dissertation, Stanford University, Stanford, CA, 1993.

[166] S. Krishnan and L. V. Kale, “A parallel array abstraction for data-driven objects,” in *Proceedings of Parallel Object-Oriented Methods and Applications Conference*, 1996, pp. 15–19.

[167] A. B. Sinha and L. V. Kale, “A load balancing strategy for prioritized execution of tasks,” in *Proceedings of International Symposium on Parallel Processing*, 1993, pp. 230–237.

[168] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 programs: Characterization and methodological considerations,” in *Proceedings of the International Symposium on Computer Architecture*, 1995, pp. 24–36.

[169] S. Chakrabarti and K. Yelick, “Implementing an irregular application on a distributed memory multiprocessor,” in *Proceedings of the Fourth ACM/SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 1993, pp. 169–179.

[170] J. A. Jones, “Parallelizing the phylogeny problem,” M.S. thesis, University of California, Berkeley, CA, 1994.

[171] Y.-S. Hwang, R. Das, J. Saltz, B. Brooks, and M. Hodoscek, “Parallelizing molecular dynamics programs for distributed memory machines,” *IEEE Computational Science and Engineering*, vol. 2, no. 2, pp. 18–29, 1995.

[172] J. P. Singh, A. Gupta, and M. Levoy, “Parallel visualization algorithms: Performance and architectural implications,” *IEEE Computer*, vol. 27, no. 7, pp. 45–56, 1994.

[173] J. Hermans and M. Carson, “CEDAR documentation.” Unpublished manual for CEDAR, 1985.

[174] J.H. Saltz et al., “A manual for the CHAOS runtime library,” Department of Computer Science, University of Maryland, Tech. Rep. CS-TK-3437, 1995.

[175] L. V. Kale, M. Bhandarkar, R. Brunner, N. Krawetz, J. Phillips, and A. Shinozaki, “NAMD: A case study in multilingual parallel programming,” in *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, 1997, pp. 37–51.

[176] A. Y. Grama, V. Kumar, and A. Sameh, “Scalable parallel formulations of the Barnes-Hut method for N-body simulations,” in *Proceedings of Supercomputing Conference*, 1994, pp. 439–448.

[177] M. Warren and J. Salmon, “A parallel hashed oct-tree N-body algorithm,” in *Proceedings of Supercomputing Conference*, 1993, pp. 12–21.

[178] B. Buchberger, *Multidimensional Systems Theory*, ch. Grobner basis: an algorithmic method in polynomial ideal theory, pp. 184–232. Dordrecht: D. Reidel Publishing Company, 1985.

[179] J.-P. Vidal, “The computation of Gröbner bases on a shared memory multiprocessor,” School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-CS-90-163, 1990.

[180] E. Chu, A. George, J. Liu, and E. Ng, “Sparspak: Waterloo sparse matrix package user's guide for Sparspak-A,” Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, Tech. Rep. CS-84-36, 1984.

[181] R. Parsons and D. Quinlan, “A++/P++ array classes for architecture independent finite difference computations,” in *Proceedings of the Second Annual Object-Oriented Numerics Conference*, 1994, pp. 408–418.

[182] D. S. Henry and C. F. Joerg, “A tightly-coupled processor-network interface,” in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, 1992, pp. 111–122.

[183] C. Whitby-Strevens, “The Transputer,” in *Proceedings of 12th International Symposium on Computer Architecture*, 1985, pp. 292–300.

[184] Anant Agarwal et al., “The MIT Alewife Machine: Architecture and Performance,” in *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995, pp. 2–13.

[185] S. Reinhardt, J. Larus, and D. Wood, “Tempest and Typhoon: User-level shared memory,” in *Proceedings of the International Symposium on Computer Architecture*, 1994, pp. 325–336.

[186] D. Chase, M. Wegman, and F. Zadeck, “Analysis of pointers and structures,” in *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, 1990, pp. 296–310.

[187] J.-D. Choi, M. Burke, and P. Carini, “Efficient flow-sensative interprocedural computation of pointer-induced aliases and side effects,” in *Twentieth Symposium on Principles of Programming Languages*, 1993, pp. 232–245.

[188] W. Landi and B. Ryder, “Pointer-induced aliasing: A problem classification,” in *Symposium on Principles of Programming Languages*, 1991, pp. 93–103.

[189] L. Hendren, A. Nicolau, and J. Hummel, “Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs,” in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 1992, pp. 249–260.

[190] S. P. Masticola and B. G. Ryder, “Non-concurrency analysis,” in *Proceedings of Fourth Symposium on Principles and Practice of Parallel Programming*, 1993, pp. 129–138.

[191] M. Rinard and P. C. Diniz, “Commutativity analysis: A new analysis framework for parallelizing compilers,” in *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996, pp. 54–67.

[192] S. Krishnan and L. V. Kale, “Automating parallel runtime optimizations using post-mortem analysis,” in *Proceedings of the 10th ACM International Conference on Supercomputing*, 1996, pp. 221–228.

[193] A. B. Sinha, “Analysis of object based and message driven programs,” Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[194] A. C. Dusseau, R. H. Arpaci, and D. E. Culler, “Effective distributed scheduling of parallel workloads,” in *ACM SIGMETRICS '96 Conference on the Measurement and Modeling of Computer Systems*, 1996, pp. 25–36.

[195] A. Dusseau-Arpaci and D. Culler, “Extending proportional-share scheduling to a network of workstations,” in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1997, pp. 191–200.

[196] P. Sobalvarro, “Dynamic coscheduling (DCS),” Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, 1996.

[197] M. Buchanan and A. A. Chien, “Coordinated thread scheduling for workstation clusters under Windows NT,” in *Proceedings of the USENIX Windows NT Workshop*, 1997, pp. 47–54.

[198] D. S. Wills, “Pi: A parallel architecture interface for multi-model execution,” Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, 1990.

VITA

Vijay Karamcheti was born in Kakinada, India, on September 13, 1967.

He received a B.Tech. degree in electrical engineering from the Indian Institute of Technology, Kanpur, India, in 1988, an M.S. degree in electrical and computer engineering from the University of Texas at Austin in 1989, and his Ph.D. degree in electrical and computer engineering from the University of Illinois, Urbana-Champaign in 1998.

Since December 1992, Vijay has worked in the Concurrent Systems Architecture Group at the University of Illinois as a research assistant with Professor Andrew A. Chien. He was a recipient of the IBM Computer Sciences Cooperative Fellowship during 1995-1997.

His research interests include computer systems architecture, and parallel and distributed computation.

Upon graduation, he will join the Courant Institute of Mathematical Sciences, New York University, New York, as an Assistant Professor of Computer Science.

For further information, Vijay may be contacted via e-mail at vijayk@cs.nyu.edu.