Architectural Support and Mechanisms for Object Caching in Dynamic Multithreaded Computations

Vijay Karamcheti
Department of Computer Science
New York University
vijayk@cs.nyu.edu

Andrew A. Chien
Department of Computer Science
University of California at San Diego
achien@cs.ucsd.edu

Abstract

High-level parallel programming models supporting dynamic fine-grained threads in a global object space are becoming increasingly popular for expressing irregular applications based on sophisticated adaptive algorithms and pointer-based data structures. However, implementing these multithreaded computations on scalable parallel machines poses significant challenges, particularly with respect to object caching. Object caching techniques must be able to tolerate unresponsive processors and protocol handler occupancy delays.

This paper examines whether these challenges can be offset by leveraging responsive general-purpose communication architectural features (such as remote memory access and atomic operations), possibly compensating for the lack of more sophisticated hardware primitives by relying upon increased involvement of the run-time system and the compiler. A detailed performance analysis of four irregular applications, using the Illinois Concert System on the Cray T3D and the SGI Origin 2000, finds that existing software distributed shared memory (DSM) systems are capable of delivering good performance only in the presence of a high level of responsive communication architecture support (specifically, support for remote atomic operations). Recognizing that this situation stems from the synchronous request-reply nature of DSM protocols, we present a composable object caching framework, called *view caching*, which exploits knowledge of application data access semantics to construct custom protocols that require reduced processor synchronization. View caching protocols are more tolerant to responsiveness and occupancy delays and are able to exploit even lower-level responsive communication primitives (such as non-atomic remote memory accesses) for a performance benefit.

1 Introduction

Irregular parallel computations are increasingly being encountered in applications such as molecular dynamics, particle simulations, and visualization, spanning a number of engineering and scientific domains. State-of-the-art techniques for these computations rely on sophisticated adaptive [3] and tree-based algorithms [2, 18], and pointer-based data structures such as linked lists, trees, and sparse matrices. High-level programming models supporting dynamic thread creation and multithreading [20, 5, 45, 28, 16] enable convenient expression of such computations by relieving the programmer from having to manage details of thread and data structuring. These models, exemplified by several concurrent object-oriented [19, 55, 11] and message-driven [28] languages, permit application concurrency to be expressed in terms of arbitrary user-defined computation units and simplify implementation of pointer-based data structures by providing a global name space.

However implementing these computations on scalable parallel machines is challenging because of the irregular thread structure and unpredictable access pattern inherent in application algorithms and data structures. One of the key challenges is efficient object caching, which must be tolerant to both spatial and

temporal irregularity situations arising from the processor interleaving inherent in multithreaded computations. Spatial irregularity arises from unbalanced thread and data access distribution across the processors, and results in *occupancy* delays where coherence requests encounter contention for processing resources on "hotspot" nodes. Temporal irregularity arises when communication operations between processors are unsynchronized with local computation, and results in *responsiveness* delays where coherence request processing is deferred until the completion of ongoing computation.

Although custom hardware support can be provided for object-level caching, our interest is in supporting object caching on general-purpose scalable parallel machines built using commodity processing node and network technology. In these environments, software distributed shared memory (DSM) systems are obvious candidates for implementing object caching. Unfortunately, most existing DSM systems are poorly equipped to handle the challenges of dynamic multithreaded computations, particularly tolerance to responsiveness delays. Most DSM systems rely upon synchronous (request-reply) operations, modeled after hardware cache-coherence protocols, and are implemented using message coroutining. While such protocols yield good performance with responsive message processing (as with dedicated hardware cache controllers), they result in large idle times when that is not the case. Even state-of-the-art DSM implementations [32, 4, 47, 26], which rely on weak consistency models to reduce coherence traffic by deferring it to synchronization points, retain the synchronous (request-reply) nature of hardware cache-coherence protocols, and require responsive processing for performance.

In this paper, we examine if these shortcomings of existing DSM systems can be alleviated by leveraging responsive general-purpose communication architectural features (such as remote memory access and atomic operations), possibly compensating for the lack of more sophisticated hardware primitives by relying upon increased involvement of the run-time system and the compiler. In contrast to custom hardware solutions, our approach is motivated by the observation that such general-purpose architectural support reflects a better cost-performance choice in the design space of parallel machines, and can additionally track advances in uniprocessor and networking technology.

We first evaluate the extent to which object caching implementations on top of existing DSM systems can directly (i.e., without interface changes) utilize responsive communication architecture features to offset responsiveness and occupancy challenges. To enable qualitative as well as quantitative characterizations, we define four generic communication interfaces—MSG, PUT/GET, LD/ST, and CC-LD/ST—and emulate them on two target platforms, the Cray T3D and the SGI Cray Origin 2000. These interfaces represent discrete design points on the spectrum from pure message-passing parallel machines (such as the CM-5), to those that provide remote memory access primitives (such as the T3D), to hardware cache-coherent shared memory machines (such as the Origin 2000). Based on a study of four irregular parallel applications using the Illinois Concert System on the T3D and the Origin, we find that although additional hardware support can overcome the effects of responsiveness and occupancy delays, a high level of hardware support is required (specifically, the presence of remote atomic operations); existing DSM systems are unable to exploit lower-level responsive primitives, such as non-atomic remote memory accesses.

Recognizing that these performance limitations stem from the use of synchronous protocols, we then describe a composable object caching framework, called *view caching*, which exploits knowledge of application data-access semantics to construct latency-tolerant coherence protocols that require reduced processor synchronization. View caching protocols are more tolerant of responsiveness and occupancy delays, and are better equipped to exploit varying levels of communication architecture support; the protocols both improve performance for a given level of hardware support, and compensate for the lack of a richer set of hardware primitives. For each of our applications, across the spectrum of communication interfaces, view caching protocols enable a high-level dynamic multithreading expression of each application to achieve performance comparable to the best reported in literature using low-level programming approaches.

The rest of this paper is organized as follows. Section 2 presents the concrete programming and execu-

tion context for this research, and introduces the application suite. Section 3 defines four generic communication architecture interfaces. In Section 4, we evaluate the extent to which the performance challenges of object caching in dynamic computations can be addressed by combining existing DSM systems with higher levels of communication architecture support. The results in this section motivate our view caching framework for constructing latency-tolerant custom protocols. Section 5 describes the design, implementation, and evaluation of the framework. Section 6 discusses related work, and we conclude in Section 7.

2 Background

This section describes the programming and execution context for the research described in this paper. Sections 2.1 and 2.2 present the programming and computation model of a specific class of dynamic multithreaded computations—fine-grained concurrent object-oriented languages, and then briefly describe the Illinois Concert System, an execution platform for such languages. Section 2.3 presents the performance challenges of object caching for such computations, and Section 2.4 introduces the four irregular applications used in this study.

2.1 Programming and Computation Model

Irregular application programs (such as molecular dynamics, particle simulations, adaptive mesh refinement, etc.) are characterized by dynamically created irregular units of work and unpredictable data access patterns resulting from use of dynamic pointer-based structures [3, 2, 18]. These characteristics are poorly supported by the three predominant parallel programming models—data parallel, message passing, and shared memory. Using these models, the programmer has to explicitly deal with the dynamic concurrency structure, irregular data locality, and program synchronization. In contrast, programming models based on a dynamic thread pool operating against shared data simplify the expression of irregular parallel computations. The *shared name space* allows programmers to build sophisticated distributed data structures without explicit name management, and *dynamic thread creation* frees programmers from explicit thread management and synchronization, allowing the irregular concurrency to be expressed in a non-binding manner that can later be adapted by the implementation to available parallelism. Such high-level programming models are supported by several multithreading [20, 5, 45, 28, 16], concurrent object-oriented [19, 12, 55, 11], and message-driven [28] languages.

In this work, we consider a specific fine-grained concurrent object-oriented language, Illinois Concert C++ (ICC++) [11], which expresses computation as interactions among a set of autonomous communicating entities (objects). ICC++ extends C++ with concurrent statements, concurrent objects, and multiaccess object collections. *Concurrent statements* express non-binding concurrency by specifying a partial order among program statements (typically, method invocations) and are declared by annotating standard blocks and loops with the `conc` keyword: statements within a `conc` block create logical threads that can be executed concurrently as long as local data dependences are preserved. *Concurrent objects* provide a consistent interface in a concurrent environment and are implicitly supported by requiring that method invocations on an object execute as if they had exclusive access. *Multiaccess object collections* support large-scale concurrency by defining an indexable set of element objects, each of which are aware of the collection. ICC++ extends the C++ array syntax to support collections, providing predefined methods which allow collections to implement an aggregate behavior and interface.

Efficient execution of the above programming model on scalable parallel machines requires mapping the logical threads across the nodes of the machine and orchestrating their data accesses. The underlying implementation maps logical threads into physical threads and interleaves processor resources among the latter (hence the name, dynamic multithreaded execution). Threads execute, by default, on the processor where

the target object (of the corresponding method) resides, although more sophisticated scheduling policies can be implemented if required. Threads access data objects in a location-independent fashion: the underlying system synthesizes a global namespace by detecting accesses to remote data objects and translating them into communication operations. This paper focuses on an efficient implementation of the global namespace using object caching.

2.2 Illinois Concert System

The above programming and computation models are supported by the Illinois Concert system which provides an optimizing compiler [42] and a high-performance run-time system to execute ICC++ programs. The compiler generates C code which links with a library of run-time primitives to produce the executable.

The Concert compiler resolves interprocedural control and data flow information [42], enabling it to specialize the generated code based on synchronization and communication features required by the computation. The compiler improves the efficiency of each thread body (employing aggressive optimizations such as method and object inlining [42, 15], procedure cloning [42], and access-region expansion [43]), reduces context-switch overheads (by controlling the amount of state saving across context switches), and generates code to take advantage of run-time object locality.

The Concert run-time system provides efficient primitives for messaging and thread management, as well as higher-level mechanisms to enhance object locality and thread load-balance. The messaging primitives are used to synthesize the global object name space, whose performance is improved using object locality mechanisms that dynamically cache remote objects. These object caching mechanisms form the focus of this paper, extending our previous work on robust messaging primitives, called *pull messaging* [30, 31]. The thread management and load-balancing mechanisms help to efficient interleave processing resources among multiple threads, and have been described in detail elsewhere [31, 29]. The Concert run-time system distinguishes itself from other multithreading run-time systems [20, 16] in providing mechanisms that incur significantly lower overhead (an order of magnitude better than vendor-supplied communication and thread libraries), and whose performance remains robust over the unpredictable dynamic behavior characteristic of irregular applications. Together with the compiler optimizations, these mechanisms enable efficient execution of fine-grained concurrent object-oriented programs on scalable parallel platforms without requiring programmer management of thread granularity, distributed name management, and initial data placement.

2.3 Performance Challenges: Responsiveness and Occupancy

The focus of this paper is on efficient object caching in the presence of two characteristics of dynamic multithreaded computations—*irregular thread structure*, and *unpredictable data access*. Irregular thread structure is a consequence of dynamic thread cration to focus effort where most beneficial, and includes irregularity in work distribution, granularity, and execution priority. Unpredictable data access arises from the fact that shared object access is intertwined with the computation and may evolve as a result of it, producing access patterns that are both unpredictable and unbalanced. The consequence of these characteristics is that object caching mechanisms need to deliver performance that is robust over both spatial and temporal irregularity situations arising from the processor interleaving inherent in dynamic multithreaded computations (see Figure 1 (left)). Spatial irregularity arises from unbalanced thread and data access distribution across the processors, and produces *occupancy* delays where coherence requests encounter contention for processing resources on "hotspot" nodes. Temporal irregularity arises when communication operations are unsynchronized with local computation, and results in arbitrary *responsiveness* delays where coherence request processing is deferred until the completion of ongoing computation.

Unfortunately, most existing software distributed shared memory (DSM) systems, which can be used for

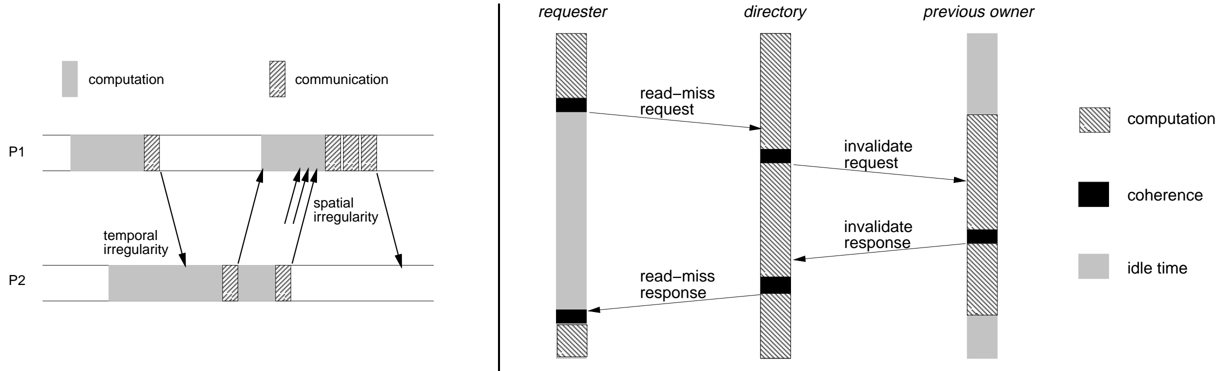

Figure 1: Spatial and temporal irregularities in implementations of dynamic multithreaded computations (left), and their impact on idle time incurred by traditional software DSM protocols (right).

object caching, are poorly equipped to handle these challenges, particularly tolerance to responsiveness delays. Most DSM systems rely upon synchronous (request-reply) operations, modeled after hardware cache-coherence protocols, and are implemented using message coroutining. While such protocols yield good performance with responsive message processing (as with dedicated hardware cache controllers), they result in large idle times when that is not the case. Even state-of-the-art DSM implementations [32, 4, 47, 26], which 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 1 (right) demonstrates this behavior in the context of cache miss processing of a read request using an invalidation-based protocol. The round-trip protocol involving the requester, the directory, and a previous owner degrades in performance because of responsiveness and occupancy delays, which increase the time a processor idles awaiting completion of a coherence operation.[1]

One possible approach for addressing the above performance challenges is to incorporate custom hardware support for object-level caching. We have adopted a different approach to address these challenges. Our approach strives to exploit the widespread availability of general-purpose responsive communication architectures such as PUT/GET, LD/ST, and CC-LD/ST, possibly with increased involvement from the run-time system and the compiler. Our approach is motivated by the observation that such general-purpose architectural support reflects a better cost-performance choice in the design space of parallel machines, and can additionally track advances in uniprocessor and networking technology.

2.4 Application Suite

Table 1 lists the applications used in this study. Each application exhibits irregularity in data structure, parallel control structure, concurrency, or all three. All of these applications have previously been implemented using either hardware support for shared memory or with explicit control of object replication and coherence. At run-time, each application exhibits wide variations in thread granularity and irregular data access patterns. The structure of each application is described below.

[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. An alternate approach—improving responsiveness by increasing the frequency of polls—suffers from the limitation that their automatic placement by the compiler requires knowledge of the program execution profile so as to offset polling overheads.

APPLICATION	DESCRIPTION	INPUT
IC-Cedar	protein dynamics [56] simulates protein macromolecular dynamics.	myoglobin
Radiosity	computer graphics [54] computes illumination using the hierarchical radiosity method.	room
Gröbner	computational algebra [10] computes Gröbner basis.	ae-4
Phylogeny	evolutionary history [27] determines the evolutionary history of a set of species.	prim.40

Table 1: The irregular applications suite.

IC-Cedar IC-Cedar is a parallel protein molecular dynamics program that models the motion of protein atoms in a solvent using Newton's equations of motion. Computation proceeds in iterations with four phases: neighbor list calculation, force calculation, motion integration, and coordinate correction. The neighbor list calculation produces for each atom, a collection of interacting atoms that are within a spatial cutoff distance. The force calculation phase dominates the execution time and computes forces incident on an atom because of its neighboring atoms. Motion integration uses these forces to update atom positions, and coordinate correction ensures validity of bond length constraints. Parallelism results from data-parallel operations over atoms. Object caching is used to cache remote atoms within an iteration; while accesses exhibit spatial locality, the access patterns are irregular because of the nonuniform spatial distribution of atoms.

Radiosity The radiosity method computes global illumination in a scene made up of diffusely reflecting surfaces. Computation proceeds in iterations, computing the radiosity of a patch as a linear combination of radiosities of patches on its interaction list (initially everyone), subdividing it or the other patches hierarchically as required to ensure accuracy. Radiosity computation requires precomputation of each interaction's visibility and formfactor, both of which need access to potentially remote patch state and rely on object caching for efficiency. The iterations terminate when the change in total radiosity falls below a threshold. The application exploits parallelism across all input patches, across child patches of a subdivided patch, and across neighbor patches stored in the interaction list. Since patches are dynamically created, object access patterns evolve during the computation.

Gröbner This application computes the Gröbner basis of a set of polynomials. The algorithm starts off with an initial basis set equal to the input set, and augments it by evaluating each pair of polynomials in a heuristic rank order. Polynomial pair evaluation can potentially add a new polynomial into the basis, which in turn generates additional polynomial pairs for evaluation. The algorithm completes when there are no unevaluated polynomial pairs. 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, relying on object caching for efficiency.

Phylogeny The Phylogeny application computes the evolutionary history of a set of species by considering the exhibited characters. Our algorithm is based on the character compatibility method, and searches for the largest compatible subset of characters by conducting a bottom-up traversal of the character lattice (depth-first and right-to-left beginning with small subsets and progressing to larger subsets), asking whether or not each subset is compatible. The algorithm maintains a failure set of subsets to facilitate pruning; each subset is first checked against the failure set prior to invoking an external procedure. Object caching enables these checks to be performed efficiently. Based on the result of the external procedure, either a subset is added

to the failure set, or new subsets are generated for exploration. Our parallel algorithm evaluates multiple character subsets as independent threads.

3 Communication Architectures

To precisely characterize how underlying architectural support affects the design and performance of object caching mechanisms, we define a common set of four interfaces that systematically cover the space of communication architectures encountered in most recent and announced scalable parallel architectures. These interfaces are described in terms of a generic node architecture consisting of a processor and memory unit connected to the interconnection network via a network interface (NI) unit.[2] 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 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):

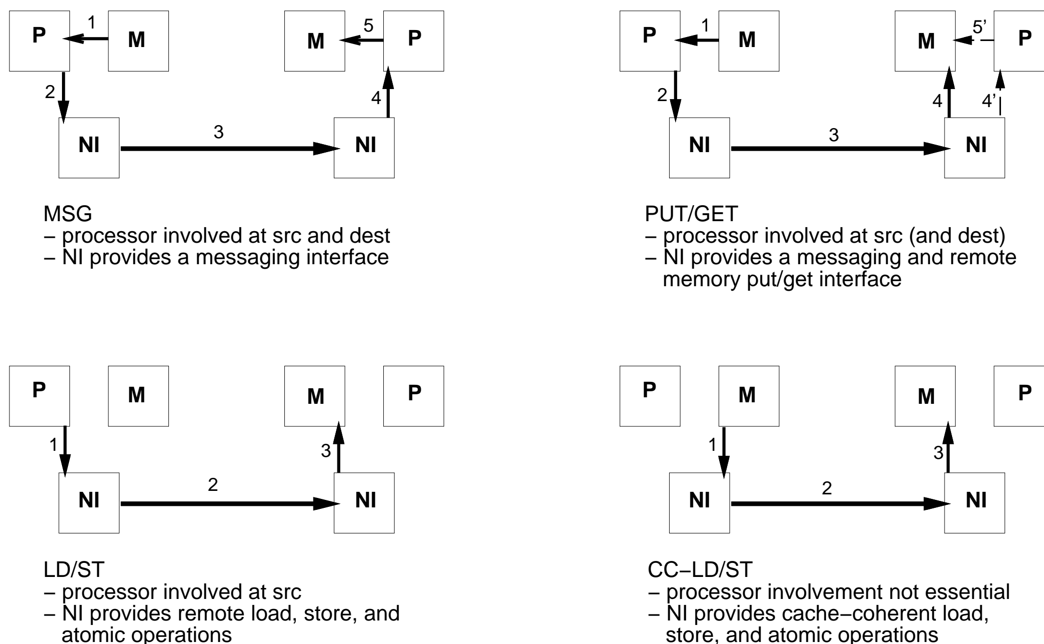

Figure 2: Four communication architecture interfaces.

- MSG supports a traditional send-receive messaging interface (as in the CM-5 [52]), 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 supports a remote memory access interface that allows a compute processor to directly write to (put) and read from (get) remote memory. Such support is present on the Princeton SHRIMP [6], the DEC Memory Channel [17], the HP Hamlyn [8], and versions of the IBM SP-x [36]. On such organizations, data transfer alone can be achieved without remote processor involvement, which is however, still required for atomic operations (shown by dashed lines).

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

- LD/ST also supports a remote memory access interface but 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 [14] and T3E [49]. 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 supports cache-coherent loads, stores, and atomic operations against remote memory, reducing effective access costs. In addition, 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 organization can model an I/O-mapped network interface processor as in the Myrinet network [7], or a hardware cache-coherent DSM multiprocessor with a custom protocol processor [38, 24, 35, 37].

The above interfaces represent discrete design points on the continuum between distributed-memory machines and a completely hardware-assisted cache-coherent shared-memory machine, and provide a platform-independent architectural context for designing caching mechanisms. However, the performance characteristics of these designs can only be quantified on a real parallel machine that supports a given interface. To avoid cross-platform performance comparisons, we consider two target platforms, the Cray T3D and the SGI Cray Origin 2000, each of which permits emulation of at least two interfaces. 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. Emulations are simply a matter of assuming restricted capabilities from the underlying hardware; for example, the PUT/GET interface can be emulated on the Cray T3D disallowing software layers from using operations such as atomic swap and fetch-and-increment that are, in fact, supported by the hardware. Emulating the same interface on multiple platforms enables us to factor out any quantitative artifacts (associated with a particular platform) and focus instead on the qualitative differences among interfaces.

4 Architectural Support for Object Caching

To understand how responsive communication architecture support impact object caching performance, we first describe the interface and implementation of an efficient object shared memory system, the C Region Library (CRL), and then, characterize the application performance for different levels of architectural support.

4.1 The C Region Library (CRL)

The C Region Library (CRL) [26] is an efficient implementation of *object consistency*, a state-of-the-art weak consistency model for object sharing. Object consistency 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 systems in literature implementing object consistency include Midway [4], SAM [47], and CRL [26]. All these systems demarcate the region in which a shared data object is accessed using program 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, an object-consistent DSM system guarantees that all updates are visible at the start of method execution.

Table 2 summarizes the CRL interface, which is used internally within the Concert run-time system to create and access shared objects. 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,

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

Table 2: Summary of the object-consistent CRL interface.

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, ensuring 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.

4.2 CRL Implementations

Renaming and coherence actions in CRL are implemented using a coroutining approach. Renaming involves a hash-based lookup scheme to resolve mappings between global region identifiers and the corresponding local objects. Coherence actions rely on 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 result in requests 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 implementation can take advantage of different levels of communication architectural support as described below (see Figure 3):

MSG. The CRL implementation for the MSG interface uses messages for implementing coherence actions. The underlying messaging infrastructure is based upon the Fast Messages [41] library, which implements an active-messages [53] like interface, and incorporates robust pull messaging [31] mechanisms which are virtually insensitive to output contention. Message handlers contain 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.

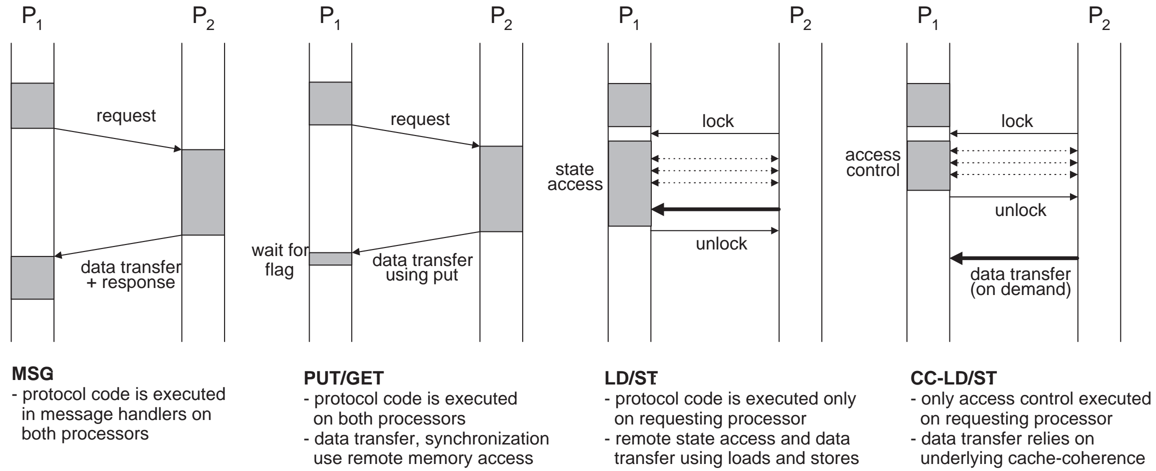

Figure 3: Four implementations of the CRL interface corresponding to various communication architectures shown in the context of a read miss. Shaded boxes represent coherence handler, solid lines represent protocol messages, bold lines represent data transfer, and dashed lines represent remote access of protocol state.

PUT/GET. The CRL implementation for the PUT/GET interface retains essentially the same structure as the MSG implementation with 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 [34, 23] 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 where 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 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 implementations 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 the improvement comes at the cost of increased overhead in protocol code due to nonlocal state accesses.

CC-LD/ST. The CRL implementation on the CC-LD/ST interface builds upon hardware cache-coherence support which maintains cache-line consistency 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}` 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.

	Cray T3D			SGI Origin 2000		
VERSION	LATENCY	OVERHEAD	OCCUPANCY	LATENCY	OVERHEAD	OCCUPANCY
Read clean copy (8 bytes)						
MSG	49.5	[24.6, 15.6, –]	[24.6, 15.6, –]	17.1	[10.8, 6.2, –]	[10.8, 6.2, –]
PUT/GET	26.3	[8.7, 10.1, –]	[8.7, 10.1, –]	15.1	[7.0, 7.1, –]	[7.0, 7.1, –]
LD/ST	28.5	[28.5, –, –]	[0.1, 0.1, –]	4.1	[4.1, –, –]	[0.1, 0.1, –]
CC-LD/ST				1.6	[1.6, –, –]	[0.1, 0.1, –]
Write with 1 invalidate (8 bytes)						
MSG	92.1	[22.2, 46.0, 12.7]	[22.2, 46.0, 12.7]	36.2	[15.2, 12.6, 8.7]	[15.2, 12.6, 8.7]
PUT/GET	60.2	[5.9, 23.8, 14.2]	[5.9, 23.8, 14.2]	31.6	[6.0, 13.6, 10.7]	[6.0, 13.6, 10.7]
LD/ST	74.3	[74.3, –, –]	[0.1, 0.1, 0.1]	10.1	[10.1, –, –]	[0.1, 0.1, 0.1]
CC-LD/ST				2.2	[2.2, –, –]	[0.1, 0.1, 0.1]

Table 3: Primitive object sharing operations on the Cray T3D and SGI Origin 2000 (all costs in μs). Overhead and occupancy costs are split into [requester, directory, remote node] components.

Table 3 shows measured costs on the Cray T3D and the SGI Origin 2000 in each of these implementations for two primitive operations: reading a clean 8-byte region, and writing an 8-byte region that involves the invalidation of one shared copy. 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* is the time when a node is unavailable to process coherence requests meant for *other* objects (due to end-point contention effects), and is also split into requester, directory, and remote node components. The occupancy is the same as the overhead when processor participation is needed to complete a coherence action. The following observations can be drawn from the data in the table:

- Our implementations on the four communication interfaces are competitive with the best reported in literature [26, 47, 46].
- The MSG implementation incurs the highest costs, attributable to the least hardware support, and additionally requires participation of the requester, home node, and remote processors indicating that their responsiveness is critical for performance.
- The PUT/GET implementation benefits from remote memory access support, improving costs over the MSG implementation, but still requires responsiveness for performance.
- The LD/ST implementation exhibits the benefits of remote atomic operations. The primary advantage is that only the requester is involved in any coherence action, implying that performance does not depend processor responsiveness or occupancy. On the T3D, the primitive costs are actually higher than the PUT/GET implementation because of the overheads associated with uncached remote loads; the Origin implementations do not show this behavior.
- The CC-LD/ST implementation shows the advantages of shifting most of the protocol actions into hardware. As with the LD/ST implementation, this implementation is also less sensitive to processor unresponsiveness and occupancy.

Thus, object shared memory implementations for different communication architecture interfaces exhibit several qualitative and quantitative differences. In the remainder of this section, we examine what impact these differences have on overall application performance.

4.3 Application Performance

Figures 4-7 show the performance achieved by each application on the Cray T3D and the SGI Origin 2000 for different communication interfaces. Each figure consists of speedup plots and histograms showing the breakdown of execution time into four components: *compute*, *overhead*, *coherence*, and *idle* time. Compute time is the time spent on actual computation. Overhead time reflects the cost of Concert run-time operations. Coherence time represents the overheads of `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. For the LD/ST and CC-LD/ST interfaces, these correspond to the time spent spinning on various locks. To facilitate direct comparison of bar heights, each graph shows all times normalized with respect to the fastest computation time for the given number of processors. For each application, the run-time system polls for messages on each coherence operation and on a logical thread dispatch. In addition, the compiler inserts poll statements on loop back edges.

The applications can be grouped into two categories based on whether or not they achieve different speedups on different architectural interfaces. The first category consists of IC-Cedar and Radiosity, and the second category consists of Gröbner and Phylogeny.

IC-Cedar (Figure 4) and Radiosity (Figure 5) benefit from increased levels of architectural support as shown by the improvement in performance using the LD/ST and CC-LD/ST interfaces as compared to the MSG and PUT/GET interfaces. However, these applications show negligible improvement with the PUT/GET interface as compared to the MSG interface. The primary factor limiting speedup is the idle time component of execution time, arising from the responsiveness and occupancy delays. The PUT/GET interface is not able to take advantage of its responsive data transfer capabilities because supporting the object-consistent DSM protocol on top of this interface requires remote processor involvement (see Table 3). In contrast, idle time constitutes a negligible fraction of the overall execution time for both the LD/ST and CC-LD/ST interfaces. The reduction stems from the fact that, on these interfaces, DSM protocols can be implemented without remote processor involvement because of support for remote atomic operations. The consequence is that application speedups on these interfaces are much better. The one exception is the performance of IC-Cedar on the Origin; here, speedups are limited by the overheads of software access control. Note that on the T3D, the idle time reduction on the LD/ST interface is even able to overcome the performance disadvantages of larger coherence overheads. Radiosity performance on the CC-LD/ST interface for 16 nodes on the Origin suffers because of additional locking and coherence overheads.[3] The performance results on these two applications indicate that communication architectural support must exceed a certain threshold of capability (more specifically, include remote atomic operations) before existing DSM protocols are able to exploit it for a performance benefit. This conclusion is also supported by an examination of how hybrid page-based DSM systems such as AURC [23] and Cashmere [34, 51] make use of hardware support for remote memory access.

On the other hand, the Gröbner and Phylogeny applications (Figures 6 and 7) achieve virtually the same performance on all interfaces with the exception of CC-LD/ST. These applications demonstrate that the mere availability of low overhead, responsive architectural support as in the PUT/GET and LD/ST architectures is inadequate to achieve high performance for dynamic multithreaded computations. A closer examination of Gröbner and Phylogeny shows that the poor performance results from processor idle times as well; however, the cause for these idle times are different than for IC-Cedar and Radiosity. In these applications, idle time results from the serialization between readers and writers enforced by the invalidation-based object-consistency protocol.[4] It must be noted that the application semantics themselves do not require this

[3]The CC-LD/ST version of each application assumes it is operating in a single shared address space, improving cross-processor data access costs at the cost of requiring method-level locking for correct execution. In contrast, the MSG, PUT/GET, and LD/ST versions work with an address space per processor, eliminating the need for these locks.

[4]Reducing the coherence granularity or adopting a multiple-writer protocol would not alleviate this problem because the reads

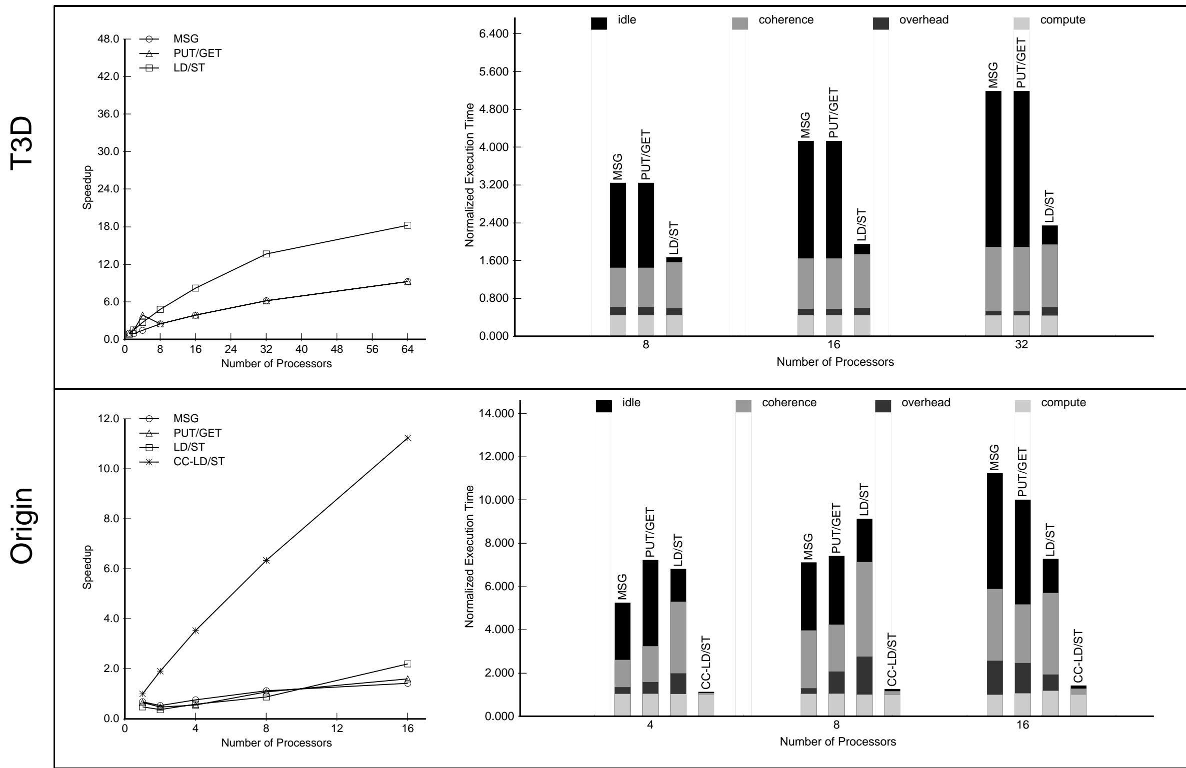

Figure 4: Performance of the IC-Cedar application using object-consistent DSM protocols.

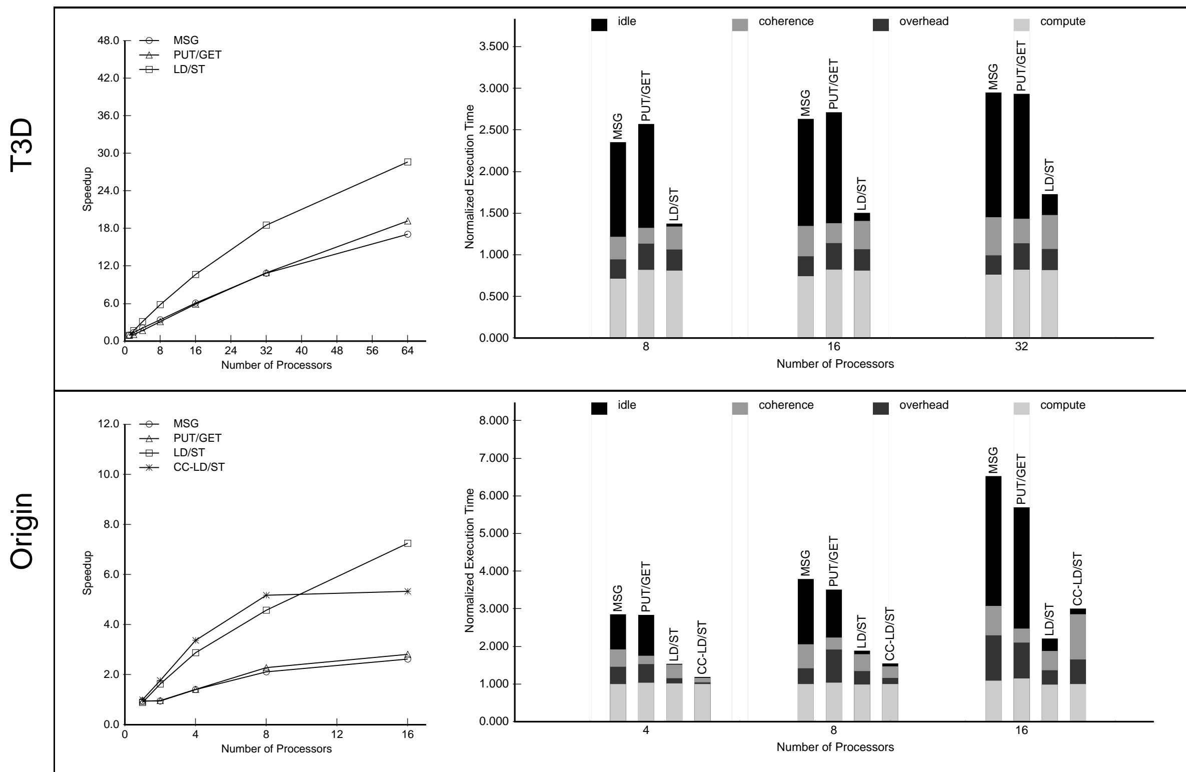

Figure 5: Performance of the Radiosity application using object-consistent DSM protocols.

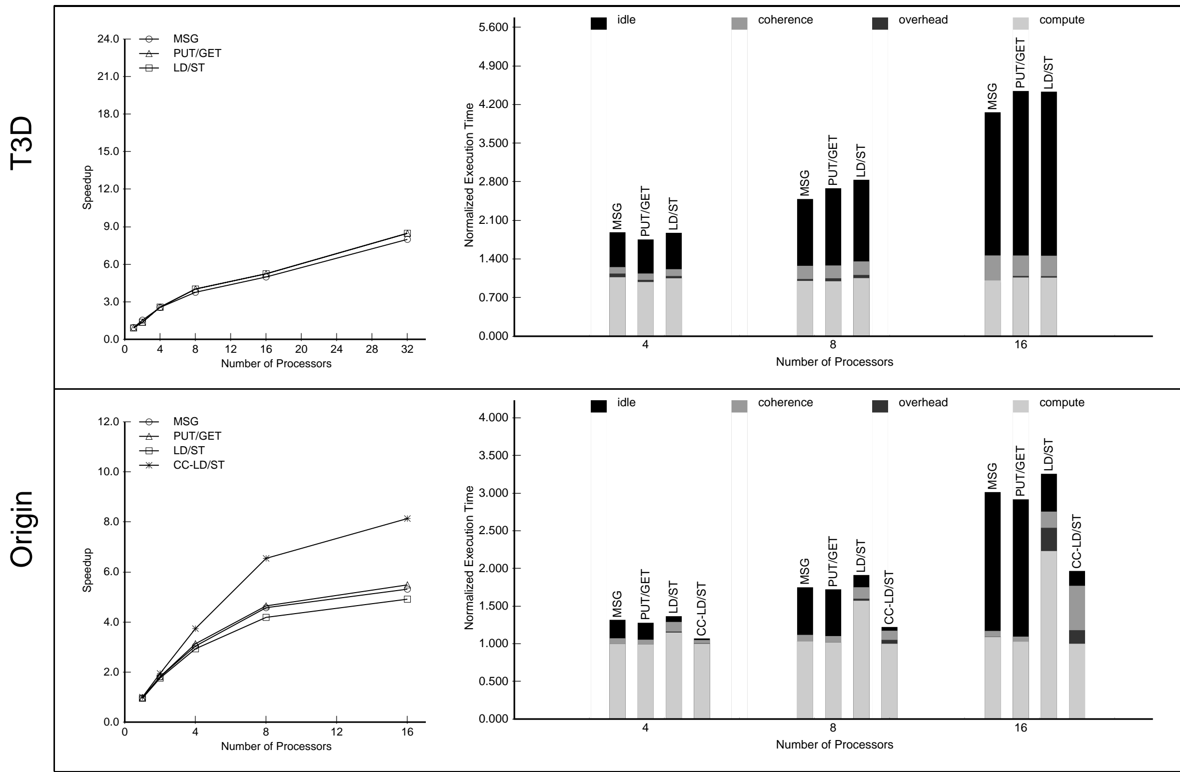

Figure 6: Performance of the Gröbner application using object-consistent DSM protocols.

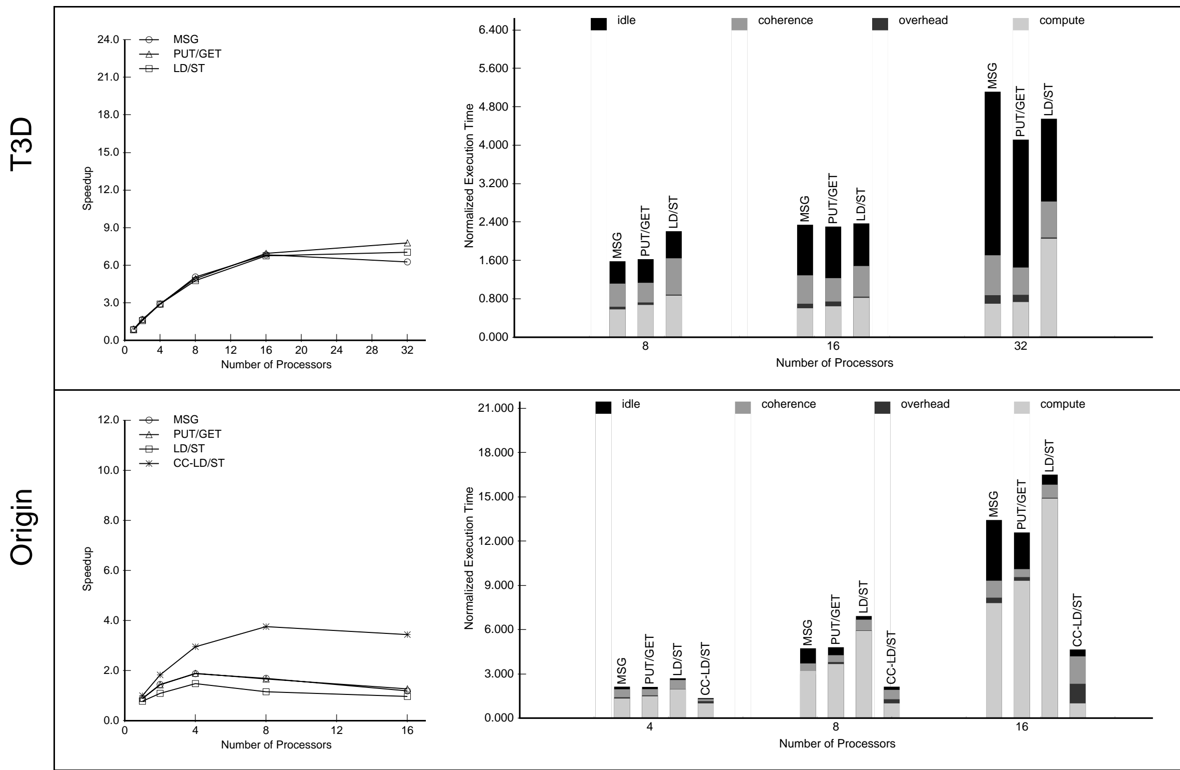

Figure 7: Performance of the Phylogeny application using object-consistent DSM protocols.

serialization: because of the underlying algorithms, basis accesses in Gröbner and failure set accesses in Phylogeny can proceed concurrently with updates. One approach for reducing these idle times is to restructure the application so that shared structures are locked at smaller granularity. In addition to the fact that such restructuring requires programmer involvement, shrinking the lock granularity increases run-time system overheads that outweigh any potential benefits.

4.4 Summary

Our application studies show that just the availability of low-overhead responsive communication architecture support is inadequate to improve object caching performance using existing DSM protocols in dynamic multithreaded computations for two reasons. First, because existing DSM protocols exhibit different sensitivity to responsiveness and occupancy as a function of the underlying interface, a relatively high level of communication architecture support (specifically, the availability of remote atomic operations as in the LD/ST interface) is required to yield performance benefits. Existing protocols are unable to utilize interfaces such as PUT/GET for a performance advantage. Second, existing protocols, even those supporting weak-consistency models, provide more restrictive semantics than may be required by the application.

5 View Caching: Latency-Tolerant Coherence

The high level of hardware support required by existing DSM protocols and their mismatch with application semantics motivates *view caching*, a framework for the construction of latency-tolerant custom coherence protocols that require less synchronization among participating processors, and hence are more tolerant to unresponsive processors and unbalanced sharing. Because of the reduced synchronization, these protocols deliver good caching performance for dynamic computations even with reduced levels of architectural support. We present in turn, the rationale behind view caching, the view caching framework, and its implementation and performance.

5.1 Rationale

Existing DSM techniques utilize synchronous (request-reply) protocols because they initiate coherence actions at the level of accesses to memory state *without utilizing any higher-level semantic information about the access*. Consequently, most implementations (i) conservatively define object copies to be consistent only if they reflect identical state, (ii) maintain globally accessible information (the *directory*) about object state and its copies, (iii) query this information prior to satisfying an access request, and (iv) conservatively invalidate or update conflicting copies before granting access. While such coupling of activities across different processors 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.

Our solution exploits application knowledge, either provided by a programmer or inferred by the compiler, to supply information both about what it means for object copies to be consistent, and how object coherence is implemented. The application knowledge helps relax the consistency requirements (as compared to object consistency) by dictating the required visibility of updates. In addition, such knowledge provides global information about the states of object copies, enabling construction of optimized protocols that do not need to query the directory. In our programming context, custom protocols can be naturally attached with methods and refer to object "views", the subset of object state accessed by a method. Custom

and writes are to the same locations.

protocols can broadly be classified into three categories, distinguished by how the consistency of the corresponding view 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.

5.2 View Caching Framework

The view caching framework defines composable primitives for building synchronous, asynchronous, and detached custom protocols that exploit application knowledge of data access semantics to reduce the required coherence actions. Unlike conventional software DSM systems which apply a single monolithic protocol for all accesses, view caching distinguishes among three components: `access-grant` refers to the processing required to satisfy the requested access, including revoking previously granted access, `access-revoke` disables further use of a cached copy, and `data-transfer` transfers object state between processors. Figure 8 shows these protocol components in the context of handling a write miss. The write miss results in a request to the home node, which in turn sends an invalidation request to a remote cache node holding a shared copy. The remote node acknowledges the invalidation request, either immediately or when it is safe to revoke access. The home node then changes directory state to reflect the new owner and transfers object state to the original requester.

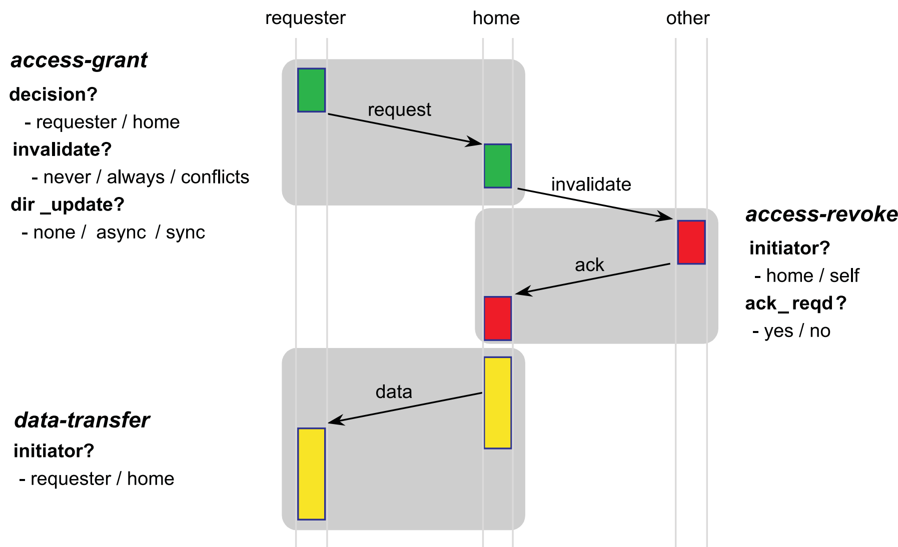

Figure 8: View caching protocols are constructed by composing customized implementations of three components: `access-grant`, `access-revoke`, and `data-transfer`. Components are customized by specifying values for a set of parameters to select from among a predefined set of implementations.

Custom protocols are built from the view caching primitives by assembling customized implementations of the three 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, the compiler

selects from a predefined set of component implementations by specifying values for a small set of parameters:

- `access-grant` is customized in terms of three parameters: decision?, invalidate?, and dir_update?. The decision? parameter controls whether the access-granting decision is performed on the requester (in detached and asynchronous protocols), or at the home node. The invalidate? parameter controls whether invalidation of existing copies is unnecessary (in asynchronous and detached protocols), always required (when it can be inferred that current access does conflict with all existing copies), or required only on conflicts (in synchronous protocols). The dir_update? parameter controls whether directory state update as part of granting the access request is unnecessary (in detached protocols), can be done asynchronously (in asynchronous protocols), or requires an acknowledgement.
- `access-revoke` is customized using two parameters: initiator? and ack_reqd?. The initiator? parameter determines whether access to a cached copy is revoked voluntarily (in asynchronous protocols) or on demand by the home node. In the latter case, the ack_reqd? parameter controls whether an acknowledgement is required or is unnecessary (in asynchronous protocols).
- `data-transfer` depends upon one parameter, initiator?, which determines whether the transfer is initiated by the requester (in some detached and asynchronous protocols), or requires home-node involvement. The former can benefit from advanced network support such as remote memory access.

Custom protocols are constructed by setting appropriate values for the above parameters. For example, if a program phase only consists of readers with no concurrent writer, an asynchronous custom protocol can be built which resolves `access-grant` on the requester asynchronously recording the readers (decision?= requester, invalidate?= never, dir_update?= async), sets up the copies for synchronous invalidation in `access-revoke` (initiator?= home, ack_reqd?= yes), and relies on requester-initiated `data-transfer` (initiator?= requester).

5.3 Prototype View Caching Implementation

For our prototype implementation, we instantiate the view caching framework to implement a suite of six custom coherence types which span the spectrum from synchronous to detached views (see Table 4). 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 choice of the coherence types was guided by the requirements of applications described in Section 2.4. These coherence types are described below.

1. `READ_SYNC` is 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. The custom protocol obtains 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`. Asynchronous views are invalidated on global synchronization points.

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

Table 4: Library of view coherence types, their description, and the corresponding coherence protocol components.

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 appear at the end of the phase. The custom protocol obtains a detached view, relying on requester-initiated access-grant and data-transfer, and requiring home node involvement 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. The custom protocol obtains a detached view that requires home node participation only for data transfer.
5. WRITE_SYNC is 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 4. 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 are less sensitive to processor responsiveness and large protocol handler occupancies.

Primitive operation costs Table 5 shows measured costs incurred by an asynchronous (READ_ASYNC) and a detached (READ_SELF_INV) view coherence type for the four communication architectures. We restrict our attention to the read operations since they benefit most from the custom protocols. In the tables, the READ_ASYNC and READ_SELF_INV implementations are labeled using the -ASYNC and -SELF suffixes.

The READ_ASYNC protocol incurs significantly reduced costs on the PUT/GET interface because the requester-initiated data transfer can be achieved without home node involvement. The quantitative differences between the T3D and the Origin result from differences in the hardware support for "gets". On the T3D, remote loads incur high costs (relative to remote stores), while on the Origin, remote loads actually incur fewer network round-trips than remote stores because of the underlying cache-coherence protocol.

The READ_SELF_INV protocol improves performance on both PUT/GET and LD/ST interfaces, benefiting from the complete elimination of home node involvement. As the occupancy costs show, this protocol is immune to all remote processor activity.

However, Table 5 does not show an important qualitative difference between the base and READ_ASYNC and READ_SELF_INV protocols on all architectural interfaces: the latter two protocols require participation of only the home node (if at all), even when another processor holds a modifiable copy of the object. Consequently, their asynchronous nature and reduced synchronization requirements are expected to contribute to a bigger performance gain, a fact verified by the application studies.

	Cray T3D			SGI Origin 2000		
VERSION	LATENCY	OVERHEAD	OCCUPANCY	LATENCY	OVERHEAD	OCCUPANCY
Read clean copy (8 bytes)						
MSG	49.5	[24.6, 15.6, –]	[24.6, 15.6, –]	17.1	[10.8, 6.2, –]	[10.8, 6.2, –]
MSG-ASYNC	59.8	[28.8, 15.6, –]	[28.8, 15.6, –]	16.4	[10.4, 5.5, –]	[10.4, 5.5, –]
MSG-SELF	51.7	[26.3, 16.2, –]	[26.3, 16.2, –]	16.7	[11.0, 5.2, –]	[11.0, 5.2, –]
PUT/GET	26.3	[8.7, 10.1, –]	[8.7, 10.1, –]	15.1	[7.0, 7.1, –]	[7.0, 7.1, –]
PUT/GET-ASYNC	34.6	[29.9, 7.2, –]	[29.9, 7.2, –]	3.2	[3.0, 0.1, –]	[3.0, 0.1, –]
PUT/GET-SELF	19.6	[19.6, –, –]	[19.6, –, –]	3.2	[3.0, –, –]	[3.0, –, –]
LD/ST	28.5	[28.5, –, –]	[0.1, 0.1, –]	4.1	[4.1, –, –]	[0.1, 0.1, –]
LD/ST-ASYNC	30.5	[30.5, –, –]	[0.1, 0.1, –]	2.9	[2.9, –, –]	[0.1, 0.1, –]
LD/ST-SELF	20.4	[20.4, –, –]	[0.1, –, –]	2.8	[2.8, –, –]	[0.1, –, –]
CC-LD/ST				1.6	[1.6, –, –]	[0.1, 0.1, –]

Table 5: Primitive object sharing operations for two view coherence types on the Cray T3D and SGI Origin 2000 (all costs in μs). Overhead and occupancy costs are split into [requester, directory, remote node] components.

5.4 Application Performance

5.4.1 Use of View Caching Protocols

Table 6 lists the custom protocols used in each application and identifies the enabling behavior. Custom protocols are enabled by concurrency and side-effect analysis of the program, which allows the programmer or compiler to infer, for each view, the required consistency semantics. These analyses identify which program fragments are concurrently active and the object side-effects in these phases. For this study that focuses on how custom protocols complement communication architecture support, we have assumed that a programmer provides the custom protocols. However, the application behavior described in Table 6 can also be automatically obtained by a compiler using aggressive interprocedural concurrency [39, 25] and side-effect [40, 13] analyses. These analyses identify, first, the fragments of thread code that are likely to execute concurrently, and then object views and associated protocols that satisfy the coherence requirements of these fragments.

Overall, the custom protocols enable three kinds of optimizations over the base DSM protocol:

1. As exemplified by the IC-Cedar, Radiosity, and Gröbner applications, threads can work with object views that are inconsistent at the object level, but consistent with respect to the subset of object state that is accessed by the thread.
2. As exemplified by the `WRITE_ALONE` protocol in the Phylogeny application, protocol actions can be optimized by taking into account the concurrent activities that are possible.
3. As exemplified by the use of `READ_SELF_INV` and `READ_ASYNC` protocols in Radiosity, Gröbner, and Phylogeny, a race in the application specification can be exploited to permit controlled inconsistency in object views. This has the effect of both reducing the number of invalidations as compared to the base protocol, and more importantly, permitting overlap of read accesses with updates.

Note that in addition to reducing the coherence traffic requirement of each application, the custom protocols make it possible to effectively exploit different levels of communication architecture support.

APPLICATION	WHERE APPLIED	PROTOCOL TYPE AND ENABLING BEHAVIOR
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_CONSTANT exploits application semantics that access is to patch state that will remain unchanged
	read access to patch object by other threads to retrieve patch radiosity	READ_SELF_INV exploits application semantics that because the specification allows a race between accesses and updates to patch state in the current iteration, the access need only be consistent 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 about a race between basis accesses and updates, permitting accesses to proceed with an out-of-date copy
Phylogeny	read access to failure set for checking subset property	READ_SELF_INV exploits application semantics about a race between accesses and updates to the failure set, permitting accesses to receive an inconsistent copy
	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

Table 6: Application behavior which enables use of custom view caching protocols.

5.4.2 Performance Impact of View Caching Protocols

Figures 9-12 show the application speedups and execution time breakdowns achieved using custom view-caching protocols. The performance of the base DSM protocols are also shown for comparison.

The overall trend across all communication architectures is that custom protocols significantly improve application performance. In fact, using these protocols, the dynamic multithreading expression of each application achieves performance comparable to the best reported in literature [22, 50, 10, 27] using lower-level approaches that either rely on a high level of hardware support, or explicitly manage object replication and coherence. For the Radiosity and Gröbner applications, performance on communication architectures without cache coherence actually exceeds that on the CC-LD/ST architecture; the latter requires additional locking and coherence overheads that are avoided in the former. 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 [48, 46].

The performance improvements stem from dramatic reductions in the idle time and coherence overheads. A detailed analysis of each application (measuring overheads of individual operations) shows that these reductions stem from three principal reasons:

1. idle time reduction in all applications arises from the use of asynchronous (READ_ASYNC) and detached (READ_CONSTANT, READ_SELF_INV, and WRITE_ALONE) protocols that reduce remote processor involvement and synchronization,

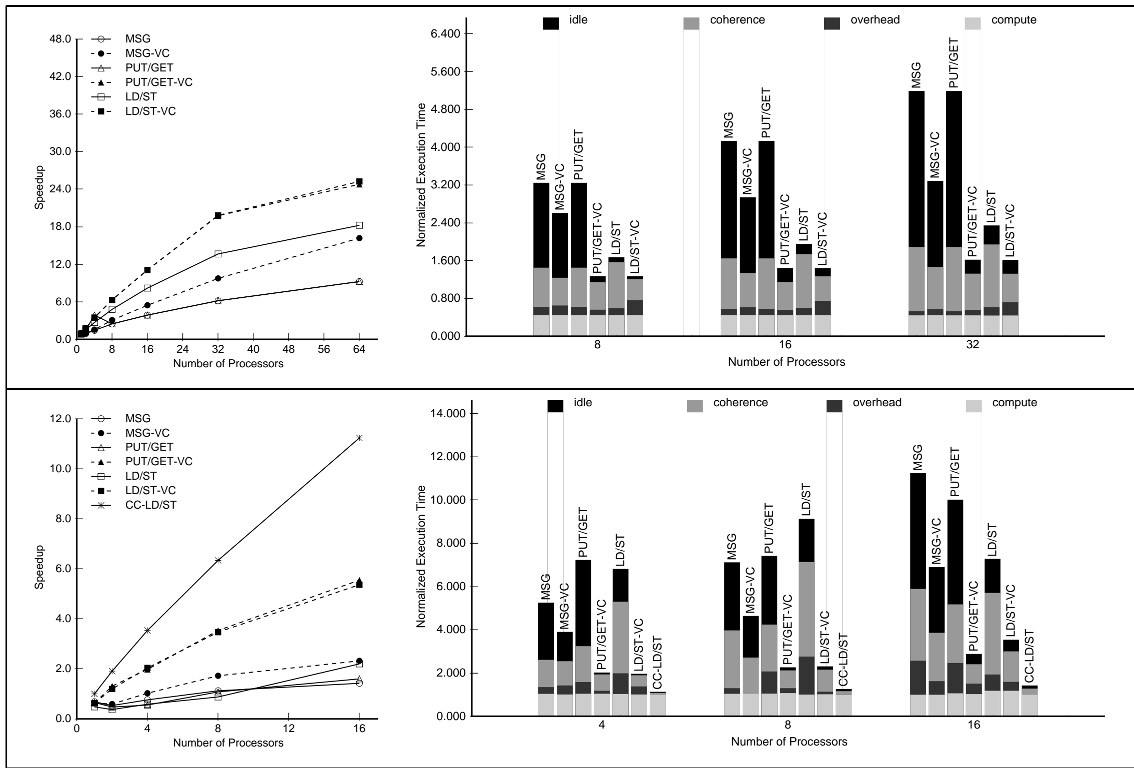

Figure 9: Performance of the IC-Cedar application using view caching protocols.

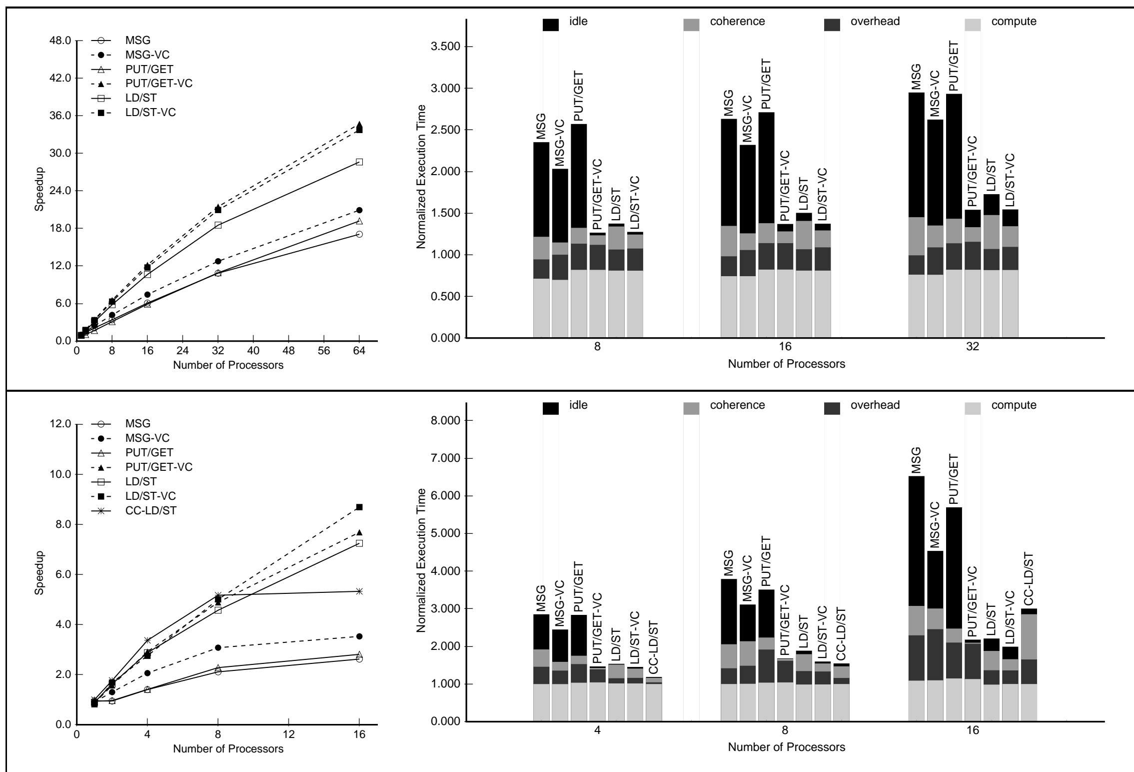

Figure 10: Performance of the Radiosity application using view caching protocols.

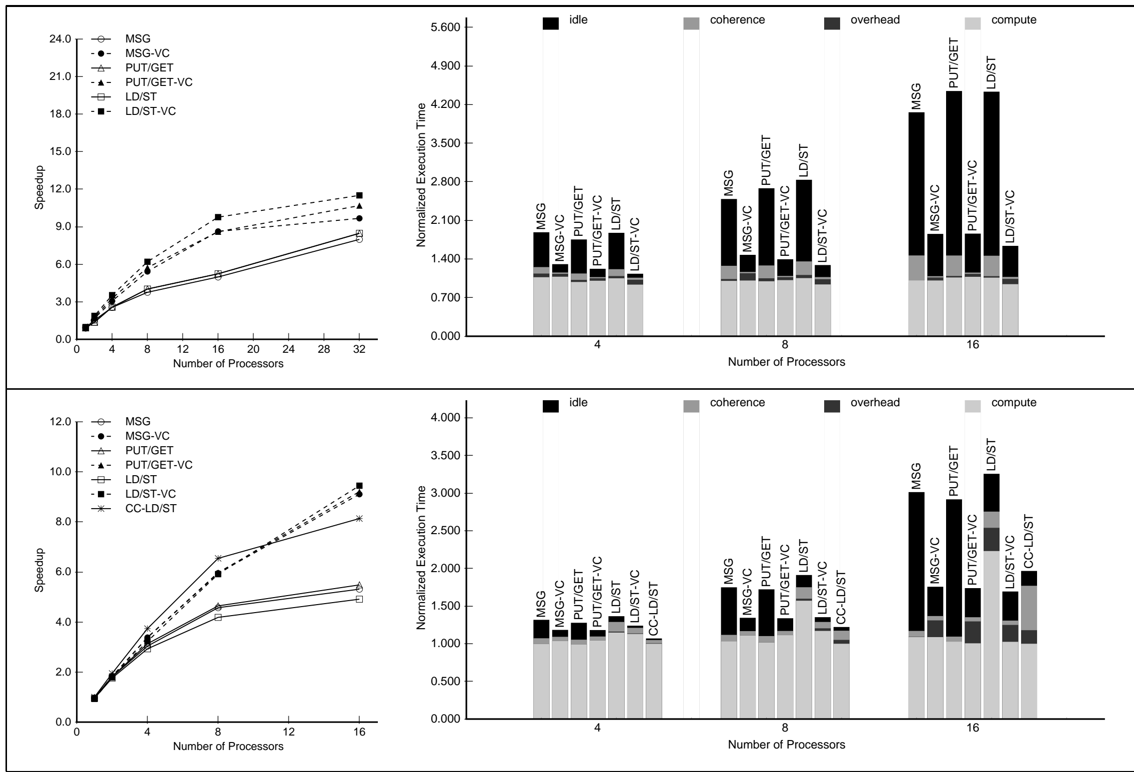

Figure 11: Performance of the Gröbner application using view caching protocols.

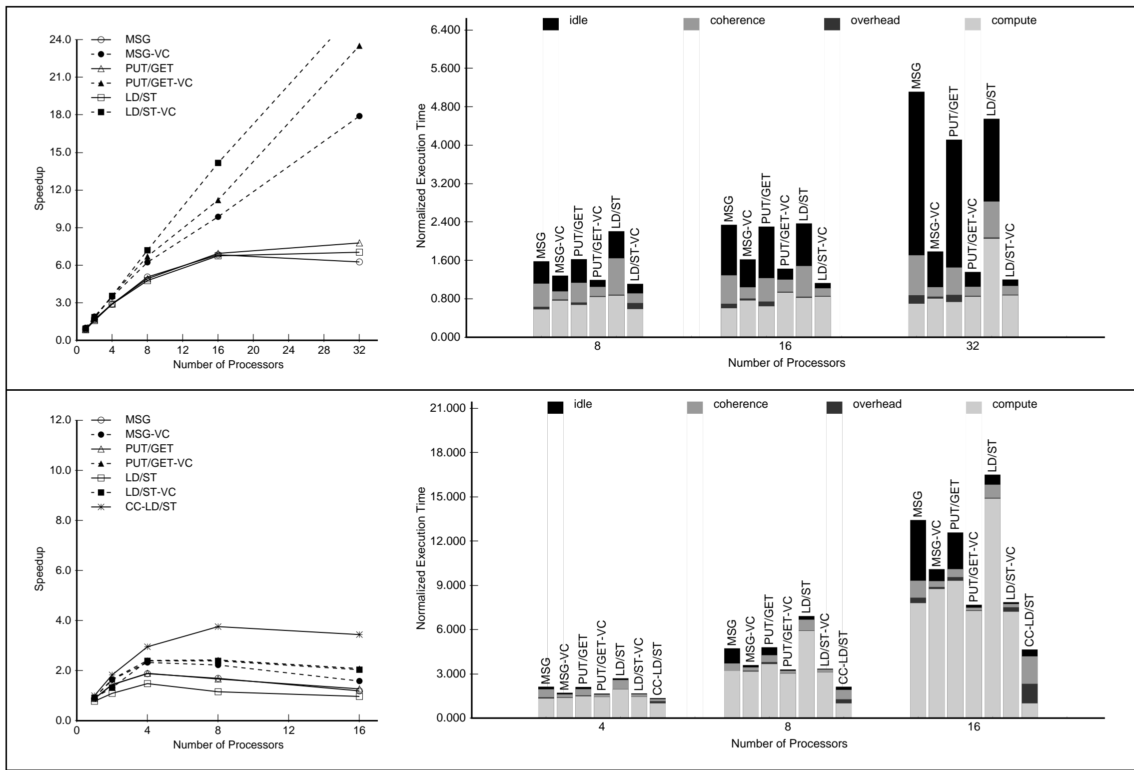

Figure 12: Performance of the Phylogeny application using view caching protocols.

2. coherence overhead reductions in Radiosity and Phylogeny stem from the use of detached views (READ_CONSTANT, READ_SELF_INV, and WRITE_ALONE) and elimination of false sharing, and
3. a large portion of the idle time reductions in Gröbner and Phylogeny are due to reduced concurrency control delays. These reductions come about because view caching protocols permit overlap of detached read views with modifications to a shared object.

In Gröbner and Phylogeny (Figures 11 and 12, performance also improves because of a reduction in the compute time component. While this may appear counter-intuitive, note that these two applications permit non-deterministic execution (for polynomial-pair evaluation and subset checking respectively) even though the final result is deterministic. In both cases, the amount of redundant work that is performed is influenced by the delay in updating the basis or the failure set. View caching protocols reduce this delay by permitting overlap of updates with ongoing reads.

Looking at how the performance impact of view caching protocols varies as a function of the communication architecture interface, we find that improvements are smallest on the MSG interface. This is explained by the fact that on the MSG interface, home node involvement is still required. The largest improvements arise for the PUT/GET interface, where requester-initiated data transfer can be implemented using hardware primitives. As the performance of IC-Cedar and Radiosity show, unlike the base DSM protocol, view caching protocols can take advantage of even reduced hardware support such as provided by the PUT/GET interface. On the LD/ST interface, applications that rely more on detached protocols (IC-Cedar) and those that benefit from read-write overlap (Gröbner and Phylogeny) show the largest improvement. On Radiosity, the magnitude of performance improvement is limited by the fact that the base protocol can also take advantage of the richer communication architecture features.

It is also interesting to observe the performance impact of view caching protocols across the range of architectures. Application measurements on both the T3D and the Origin show that custom protocols on a particular communication architecture achieve performance parity with base protocols on a more sophisticated architecture, suggesting that customizing protocols using knowledge of application semantics can compensate for a lack of hardware support.

5.5 Summary

View coherence protocols take advantage of application knowledge to decouple processor activities and reduce synchronization, improving application performance across a range of communication architectures. These performance advantages are also available on simpler communication architectures such as MSG and PUT/GET and arise from qualitative advantages of the protocols (less synchronization and processor coupling), persisting despite higher per-operation costs as on the T3D. Thus, view caching protocols can be considered as either maximizing the utility of a given level of communication architecture support, or permitting a particular level to be used in lieu of a higher level of support.

6 Discussion and Related Work

This paper has examined how general-purpose communication architecture support, which is available in several current and announced parallel machines, can be combined with a novel object caching run-time mechanism, called view caching, to deliver high performance for dynamic multithreaded computations.

The study originated with the observation that object caching in dynamic multithreaded computations is poorly supported using existing software DSM systems on parallel platforms that do not provide hardware cache coherence. One approach for addressing these shortcomings is to include custom hardware support for object caching, potentially building on hardware support for cache-coherence in flat address spaces [38, 1,

44, 24]. Instead, we have adopted an approach that strives to leverage general-purpose architectural support, possibly compensating for the lack of hardware primitives by relying upon increased involvement of the run-time system and the compiler. Our approach is motivated by the observation that such general-purpose architectural support reflects a better cost-performance choice in the design space of parallel machines, and can additionally track advances in uniprocessor and networking technology.

Supporting shared memory in software, relying only on general-purpose hardware support is an area that has been studied by several researchers. Related previous research can be classified into three broad categories: *weak consistency models*, *application-specific protocol customization*, and *hardware-augmented DSM systems*.

Weak consistency models Several researchers have proposed weaker consistency models in the context of both page-based and object-based DSM systems. In the first category are consistency models such as release consistency, lazy release consistency [32], and message-driven release consistency [33], while examples of the latter include several variants of entry consistency used in the Midway [4], SAM [47], and CRL [26]. These models reduce required coherence traffic by deferring all coherence actions to synchronization points, but still rely on request-reply protocols mandating prompt servicing of coherence requests. As our evaluation of object-consistent DSM in Section 4 demonstrates, while these protocols yield good performance for regular applications, their use of synchronous request-reply protocols shows high sensitivity to processor unresponsiveness and protocol handler occupancy in the context of dynamic multithreaded applications. The latter degrades performance on communication architectures such as MSG and PUT/GET.

Application-specific protocol customization Systems such as Munin [9] and Tempest [21], which allow application-specific customization of coherence protocols, are most similar to our view-caching approach. Munin allows custom coherence actions for shared regions by selecting among a set of predefined coherence protocols that reflect common application-sharing scenarios. The Tempest approach is more general, permitting an arbitrary user-defined handler to be invoked to service an access request. View caching is very similar to these systems, yet there are important differences. Unlike Munin which permits all accesses to an object to the same coherence policy, our approach permits customization at the granularity of use (i.e., the view). As our application results show, treating each view as an independent coherence unit provides additional opportunities for optimization. Our approach is similar to, but significantly more general than, the support for chaotic accesses in SAM [47]. An additional distinction is that view caching primitives were chosen based on their suitability for being composed into protocols with reduced synchronization requirements. Such protocols are important for efficient object caching in dynamic multithreaded computations, and are able to effectively exploit different levels of communication architecture support.

Hardware-augmented DSM systems This design of DSM protocols to complement higher levels of communication architecture support becoming available in current-day networks is most similar to the AURC [23] and Cashmere [34, 51] projects. The similarity is that both sets of systems attempt to utilize hardware support for remote memory access to complete a coherence action on the requester, avoiding home-node processing bottlenecks. As our results in Section 4 show, such approaches require support for remote atomic operations as a prerequisite and are not nearly as effective with lower levels of support as in a PUT/GET interface. The primary difference of our approach is that it provides a framework for incorporating application-specific coherence semantics to altogether avoid coherence actions on remote nodes, thereby permitting effective use of even lower levels of communication architecture support.

7 Conclusion

This paper describes a composable object caching framework, called view caching, for dynamic multi-threaded computations, which combines availability of responsive communication architecture support in current-day parallel machines with knowledge of application data-access semantics to construct custom protocols that deliver performance robust over unresponsive processors and protocol handler occupancy delays.

The view caching framework is motivated by application studies, using the Illinois Concert System on the Cray T3D and the SGI Origin, which show that existing software DSM systems require a high level of communication architecture support (remote atomic operations) for good performance; such systems are unable to take advantage of lower-level responsive primitives such as non-atomic remote memory access. In contrast, view caching protocols are better able to exploit varying levels of communication architecture support; the protocols both improve performance for a given level of hardware support, and compensate for the lack of a richer set of hardware primitives. These protocols enable high-level dynamic multithreading expressions of applications to achieve performance comparable to the best reported in literature using low-level programming approaches.

Acknowledgements

The authors would like to acknowledge John Plevyak, Julian Dolby, Xingbin Zhang, Scott Pakin, and other members of the Concurrent Systems Architecture Group for their work on various parts of the Illinois Concert System and for their assistance in developing the ideas presented here.

The research described in this paper was supported in part by DARPA Order #E313 through the US Air Force Rome Laboratory Contract F30602-96-1-0286, NSF grant MIP-92-23732, ONR grants N00014-92-J-1961 and N00014-93-1-1086, NASA grant NAG 1-613, and supercomputing resources at NCSA and the Jet Propulsion Laboratory. Support from Intel Corporation, Tandem Computers, Hewlett-Packard, and Motorola is also gratefully acknowledged. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809. Vijay Karamcheti was supported in part by an IBM Computer Sciences Fellowship.

References

[1] Anant Agarwal et al. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2–13, June 1995.

[2] J. Barnes and P. Hut. A hierarchical O(N log N) force calculation algorithm. Technical report, The Institute for Advanced Study, Princeton, New Jersey, 1986.

[3] Marsha J. Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[4] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proceedings of COMPCON 1993*, pages 528–537, March 1993.

[5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, Andrew Shaw, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of Principles and Practice of Parallel Programming*, pages 207–216, 1995.

[6] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceeding of the International Symposium on Computer Architecture*, pages 142–153, April 1994. Available from `http://www.cs.princeton.edu/shrimp/papers/isca94.paper.ps`.

[7] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet—a gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–36, February 1995. Available from `http://www.myri.com/research/publications/Hot.ps`.

[8] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An implementation of the Hamlyn sender-managed interface architecture. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 245–259, October 1996. Available from `http://www.hpl.hp.com/personal/John_Wilkes/papers/HamlynOSDI96.pdf`.

[9] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 152–164, 1991.

[10] 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*, pages 169–179, May 1993.

[11] Andrew Chien and Uday Reddy. ICC++ language definition. Concurrent Systems Architecture Group Memo, Also available from `http://www-csag.cs.uiuc.edu`, February 1995.

[12] Andrew A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs*. MIT Press, Cambridge, MA, 1993.

[13] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–66, June 1988.

[14] Cray Research, Inc., Eagan, MN. *Cray T3D System Architecture Overview*, March 1993.

[15] Julian Dolby. Automatic inline allocation of objects. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 7–17, June 1997.

[16] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *J. Parallel and Distributed Computing*, 37(1):70–82, 1996.

[17] Richard B. Gillett. Memory Channel network for PCI. *IEEE Micro*, 16(1):12–18, February 1996. Available from `http://www.computer.org/pubs/micro/web/m1gil.pdf`.

[18] L. Greengard and V Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–48, 1987.

[19] A. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer*, 5(26):39–51, May 1993.

[20] Matthew Haines, David Cronk, and Piyush Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing'94*, pages 350–359, November 1994.

[21] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A substrate for portable parallel programs. In *Compcon*, March 1995. Available from `ftp://ftp.cs.wisc.edu/wwt/compcon95_tempest.ps`.

[22] Yuan-Shin Hwang, Raja Das, Joel Saltz, Bernard Brooks, and Milan Hodošček. Parallelizing molecular dynamics programs for distributed memory machines. *IEEE Computational Science and Engineering*, 2(2):18–29, Summer 1995.

[23] Liviu Iftode, Cezary Dubnicki, Edward W. Felten, and Kai Li. Improving release-consistent shared virtual memory using automatic update. In *Proceedings of the 2nd International Symposium on High Performance Computer Architecture*, pages 14–25, February 1996.

[24] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[25] T. E. Jeremiassen and S. J. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, August 1994.

[26] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the Symposium on Operating Systems Principles*, pages 213–228, 1995.

[27] Jeff A. Jones. Parallelizing the phylogeny problem. Master's thesis, University of California, Berkeley, CA, December 1994.

[28] L. V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of OOPSLA'93*, pages 91–108, 1993.

[29] Vijay Karamcheti. *Run-Time Techniques for Dynamic Multithreaded Computations*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1998.

[30] Vijay Karamcheti and Andrew 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*, pages 298–307, 1995. Available from `http://www-csag.cs.uiuc.edu/papers/cm5-t3d-messaging.ps`.

[31] Vijay Karamcheti, John Plevyak, and Andrew A. Chien. Runtime mechanisms for efficient dynamic multithreading. *Journal of Parallel and Distributed Computing*, 37(1):21–40, 1996. Available from `http://www-csag.cs.uiuc.edu/papers/rtperf.ps`.

[32] 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*, pages 115–132, January 1994.

[33] Povl T. Koch, Robert J. Fowler, and Eric Jul. Message-driven relaxed consistency in a software distributed shared memory. In *First Symposium on Operating Systems Design and Implementation*, pages 75–85, November 1994.

[34] 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*, pages 166–177, February 1996.

[35] John Kubiatowicz and Anant Agarwal. The anatomy of a message in the Alewife multiprocessor. In *Proceedings of the International Conference on Supercomputing*, pages 195–206, July 1993. Available from `ftp://cag.lcs.mit.edu/pub/papers/anatomy.ps.Z`.

[36] James M. Kuzela. *IBM POWERparallel System - SP2, Performance Measurements*. Power Parallel Division, IBM, February 1995.

[37] Jim Laudon and Dan Lenoski. System overview of the SGI Origin 200/2000 product line. In *Proceedings of COMPCON 97*, pages 150–156, February 1997.

[38] Daniel Lenoski and et al. The Stanford DASH Multiprocessor. *IEEE Computer*, pages 63–79, Mar 1992.

[39] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *Proceedings of Fourth Symposium on Principles and Practice of Parallel Programming*, pages 129–138, May 1993.

[40] E. Myers. A precise interprocedural data flow algorithm. In *Eighth Symposium on Principles of Programming Languages*, pages 219–30, 1981.

[41] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages: Efficient, portable communication for workstation clusters and mpps. *IEEE Concurrency*, 5(2):60–73, April-June 1997. Available from `http://www-csag.cs.uiuc.edu/papers/fm-pdt.ps`.

[42] John Plevyak. *Optimization of Object-Oriented and Concurrent Programs*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1996.

[43] John Plevyak, Xingbin Zhang, and Andrew A. Chien. Obtaining sequential efficiency in concurrent object-oriented programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 311–321, January 1995.

[44] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the International Symposium on Computer Architecture*, pages 325–336, April 1994.

[45] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[46] Daniel Scales, Kourosh Gharachorloo, and Chandramohan Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of ASPLOS-VII*, pages 174–185, October 1996.

[47] Daniel J. Scales and Monica S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *First Symposium on Operating Systems Design and Implementation*, pages 101–114, 1994.

[48] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David Wood. Fine-grain access control for distributed shared memory. In *ASPLOS-VI*, pages 297–306, October 1994.

[49] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 26–36, Cambridge, Massachusetts, October 1996. Available from `http://reality.sgi.com/sls_craypark/Papers/asplos96.html`.

[50] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7):45–56, July 1994.

[51] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. CASHMERE-2L: Software coherent shared memory on a clustered remote-write network. In *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.

[52] Thinking Machines Corporation, Cambridge, MA. *The Connection Machine CM-5 Technical Summary*, October 1991.

[53] 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*, pages 256–266, 1992.

[54] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture*, pages 24–36, 1995.

[55] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. MIT Press, Cambridge, MA, 1990. ISBN 0-262-24029-7.

[56] Xingbin Zhang, Vijay Karamcheti, Tony Ng, and Andrew Chien. Optimizing COOP languages: Study of a protein dynamics program. In *IPPS'96*, pages 235–240, 1996.