Supporting High Level Programming with High Performance: The Illinois Concert System

Andrew Chien   Julian Dolby   Bishwaroop Ganguly   Vijay Karamcheti
Xingbin Zhang
Department of Computer Science
University of Illinois
Urbana, Illinois 61801
concert@red-herring.cs.uiuc.edu

Abstract

Programmers of concurrent applications are faced with a complex performance space in which data distribution and concurrency management exacerbate the difficulty of building large, complex applications. To address these challenges, the Illinois Concert system provides a global namespace, implicit concurrency control and granularity management, implicit storage management, and object-oriented programming features. These features are embodied in a language ICC++ (derived from C++) which has been used to build a number of kernels and applications.

As high level features can potentially incur overhead, the Concert system employs a range of compiler and runtime optimization techniques to efficiently support the high level programming model. The compiler techniques include type inference, inlining and specialization; and the runtime techniques include caching, prefetching and hybrid stack/heap multithreading. The effectiveness of these techniques permits the construction of complex parallel applications that are flexible, enabling convenient application modification or tuning. We present performance results for a number of application programs which attain good speedups and absolute performance.

Keywords
concurrent languages, concurrent object-oriented programming, compiler optimization, runtime systems, object-oriented optimization

1 Introduction

The increasing complexity of software concomitantly increases the importance of tools which reduce, or aid the management of, such complexity. This trend is profoundly reshaping the mainstream programming world which focuses on sequential computing, producing a large scale movement toward object-oriented languages (e.g. C++ [13], Smalltalk [15] and Java [38]) and object-based programming techniques (e.g. CORBA [30], DCOM, OLE, and a wealth of other object standards). Such techniques provide encapsulation which supports code reuse and modularity (separate design), enabling the construction of larger, more complex systems and speeding the development process. The Illinois Concert system extnds these object-oriented techniques to also manage the additional complexity inherent in parallel programming.

Parallel computing further complicates software development by introducing distribution (locality and data movement) and concurrency (parallelism, synchronization, and granularity). While careful design by the programmer is always essential, some of these issues can be dealt with automatically by a programming system. The Concert system is designed to allow programs to avoid explicit specification of choices where possible (increasing program flexibility and portability) by automating the choices in an optimizing compiler and runtime. In addition, the Concert system provides languages which allow applications to be constructed flexibly regarding key design choices (even data placement for some applications), allowing architects to reconsider and modify their choices as more application and systems requirements become clearer. Specific high-level features that Concert supports include:

- global object namespace
- flexible object-oriented programming model
- nonbinding concurrency specification
- implicit concurrency control through data abstractions
- implicit thread/object/execution granularity
- implicit storage management

Thus, Concert programmers exploit high level features to construct complex applications with diverse forms of concurrency. A high level style enables flexible programs and even implementations which can improve application performance.

However, the primary drawback of high level programming models is their perceived inefficiency compared to low-level competitors such as message passing within either C or Fortran. However, each of the high level features provided by the Illinois Concert system is efficiently implemented by exploiting techniques that are portable, general across a range of program structures, and achieve efficient execution. These techniques include compiler analyses and transformations, runtime optimizations, and often unique combinations of the two. The implementation techniques can be roughly classified as supporting each of the key high level features enumerated above:

- aggressive interprocedural analysis and optimization (flexible object-oriented programming model, implicit granularity, implicit concurrency control and nonbinding concurrency specification)
- efficient primitives for threading, communication (implicit granularity, global object namespace)
- dynamic adaptation (implicit thread/execution granularity, global object namespace)
- custom object caching, dynamic pointer alignment (global object namespace)
- concurrent garbage collection (implicit storage management)

The Concert system has been an ongoing project for over four years, and not only have we built a working system, we have demonstrated the above techniques on numerous application programs and kernels. These demonstrations have repeatedly confirmed the benefits of high-level programming constructs and aggressive implementation techniques for achieving high performance, flexible application software. Application demonstrations are used to illustrate the good speedups and high absolute performance levels achieved, and also to assess the impact of various optimizations on program performance.

1.1 Organization

The rest of this paper is structured as below. Section 2 describes the ICC++ language, focusing on its concurrency features and support for optimization. In Section 3, we present implementation strategy: the global analysis framework used by the Concert compiler (Section 3.1) and the static transformations that it enables (Section 3.2); Sections 3.3 through 3.5 detail the dynamic adaptation features of Concert and the runtime. Performance results for both sequential and parallel programs are presented in Section 4. We discuss related high-performance language systems in Section 5, and conclude in Section 6.

2 Language Support: ICC++

ICC++ is the Illinois Concert C++ language, designed to provide both high level programming and high performance execution. The language support for high level programming and its efficient implementation can be divided into four parts: a general high level model, the expression of concurrency, and concurrency control (synchronization). ICC++ addresses the managing large-scale concurrency with highly parallel object collections. We cover the salient features of ICC++ for each of these language features in order. Further details of ICC++ can be found in [17].

2.1 General High Level Features

ICC++ supports flexible, modular programming for concurrent programs in a style similar to that available for sequential programs. The basic features of the model which support this capability include:

- object-oriented programming
- global object namespace
- implicit storage management

Object oriented programming has been demonstrated as useful for improving the organization and modularity of large programs. It can also be thought of as a way of exploiting flexible granularity – in methods and data – to reduce the complexity of programming. **A global object namespace** allows data to be accessed uniformly, factoring data and task placement from the program's functional specification. **Implicit storage management** frees the programmer from the details of memory management. Long a recognized benefit in sequential programs [22, 15] and recently further popularized by Java [38], implicit storage management simplifies concurrent programs significantly, particularly those with complex distributed data structures – the Concert project's primary focus.

2.2 Expressing Concurrency

ICC++ declares concurrency by annotating standard blocks (i.e. compound statements) and loops with the `conc` keyword. Both constructs are illustrated in Figure 1.

A `conc` block defines a *partial* order, allowing concurrency between two statements except in two cases: 1) an identifier appearing in both is assigned or declared in one of them, or 2) the *second* statement contains a jump. `conc` loops extend this semantics by treating loop carried dependences as if the variable were replicated for each iteration,

```
conc {                // statement
  Foo *b = new Foo(a); // s1
  r1 = b.meth_1();     // s2
  r2 = b.meth_2();     // s3
  return (r1 + r2);    // s4
}
conc while (i < 5) {   // s5
  a->foo(i);           // s6
  i = i + 1;           // s7
}
```

Figure 1. ICC++ Concurrency Constructs

and loop-carried definitions of a given variable assigned to the next iteration's copy of it.

In Figure 1, for example, s1 must finish before s2 and 3 can start, and they both must finish before s4 can commence. In the loop example, the iterations unfold sequentially, since the loop test must wait for statement s7, but since neither s5 nor s7 need wait for s6, executions of s6 proceed concurrently as the loop unfolds.

In short, both `conc` forms preserve local data dependences, facilitating applying them to sequential programs. And both constructs indicate *non-binding* concurrency, allowing an implementation to serialize or not as it deems best for efficient execution.

2.3 Managing Concurrency

In most programs, concurrency must be constrained (i.e. synchronization is needed) to ensure correct execution. In ICC++, consistency is managed at the object (data abstraction) level: by ensuring that concurrent method invocations on an object are constrained such that intermediate object states created within a member function are not visible, the language semantics gives method invocations apparent exclusivity. Thus, the state of each object is sequentially consistent. There is no consistency guaranteed between objects, but this capability can, naturally, be used to build arbitrary multi-object synchronization structures. These semantics were chosen both to provide programming convenience and allow compiler optimization of locking overhead [9, 34, 31]. Concrete examples are provided in the next section.

2.4 Large Scale Concurrency: Collections

Large scale concurrency often requires manipulation of large groups of objects, as well as the systematic elimination of any single points of serialization, to achieve good scalability [8]. To support clean encapsulation of large scale concurrency, ICC++ incorporates collections of objects. Objects within a collection are aware of the collection, and hence can co-operate to implement an abstraction with a concurrent interface. A collection is declared as a normal class but with `[]` appended to the class name:

```
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; }
}
```

The above declaration creates two classes: `Accum` and `Accum[]` representing the elements and whole collection respectively. Note that concurrent calls to `Accum::add` and `Accum::total` on different elements may happen in parallel, and the concurrency is entirely hidden. Thus, this distributed accumulator could be substituted transparently for a sequential one.

Note the role played by the object consistency model: by hiding intermediate states, it eliminates any low-level race conditions associated with concurrent updates to a specific element's `Accum::local` (read-modify-write by +=); at the same time, multiple calls to `Accum[]::total` and `Accum::total` may proceed concurrently as they do not create intermediate states, presenting a concurrent interface.

3 Efficient Implementation Strategy

The high-level features of Concert that simplify programming also complicate implementation. The effect of object-oriented abstraction is to hide implementation details needed for efficient code – e.g. concrete types of variables and object lifetimes – beneath abstract interfaces, requiring program analysis to discover them. Additionally, the many interfaces in object-oriented code tend to break programs down into many small, dynamically-dispatched methods [3]. Two performance issues arise from this:

Small dynamic methods both increase overhead by requiring function calls and reduce the effectiveness of standard intra-procedural optimizations by giving them smaller function bodies to work on. Inlining is vital for high performance, and type information is required to make inlining possible in the face of dynamic dispatch.

Implicit storage management gives all object conceptually infinite lifetimes, increasing overhead by requiring heap allocation and garbage collection. Optimization can reduce such overhead by removing unnecessary objects.

These problems are well-studied for sequential object-oriented models; additionally, our approach to concurrency adds further challenges:

Global object namespace hides whether or not a given object is local to another object accessing it; however, if it is known to be local, much more efficient access and other collateral optimizations are possible, hence locality analysis is important.

Implicit concurrency control eliminates explicit locks for object consistency, requiring the system to ensure it. Optimization must amortize the cost of locking over the largest possible regions of code.

Non-binding concurrency leaves the system to determine when to execute sequentially and when to create parallel work; this means balancing parallel overhead against parallel speedup, and addressing load balance.

To address these challenges, the compiler first performs inter-procedural data-flow analysis to discover type information, locality information and object relationships. This information is exploited at four levels: 1) static transformation of methods and object structures to increase their granularity to amortize the cost of ensuring access and locality. Beneath that, in cooperation with the runtime system, 2) locality management supports distributed data structures and 3) light-weight thread support enables fine-grained concurrency. This all rests on 4) efficient runtime mechanisms for communication and thread scheduling.

3.1 Program Analysis

The Concert compiler implements global program analysis [32, 31] to obtain a variety of information: types of variables to resolve dynamic disptach, relative locality of objects, and container objects for storage optimizations. The analysis is *context sensitive* and adapts, in a demand-driven manner, to program structure. To prevent information loss, the analysis creates contexts (representing program environments) for differing uses of classes (e.g. polymorphic containers) and methods (e.g. differing types for a given argument at different call sites).

These contexts are created on demand when the analysis needs to distinguish some property. One analysis done is type inference, which creates contexts to distinguish type information. Method contexts are created for sets of argument types; for polymorphic containers, different class contexts are built for the containing object to differentiate the types in the field.

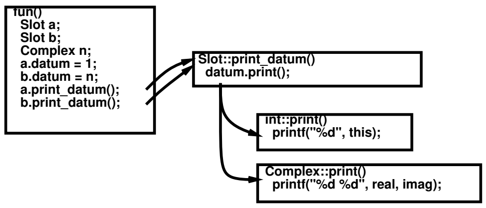

Figure 2. Analysis Pass One

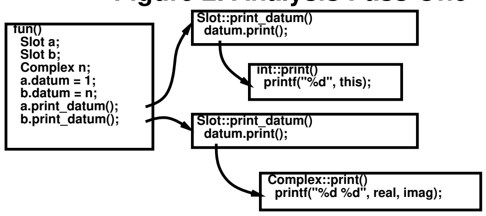

Figure 3. Analysis Pass Two

Figures 2 and 3 illustrate type analysis on a simple program fragment with polymorphic containers after a single pass of adaptive analysis. The two calls to `print_datum` have the same argument types (both are called on a `Slot`), so they share a context. Within `print_datum`, the type of `datum` is ambiguous, requiring dynamic dispatch. Since this type confusion is due to a field, the system, during the next pass of analysis, creates class contexts for `Slot` to distinguish the types of `Slot::datum`. This, in turn, causes two contexts to be created for `print_datum`, as the targets now come from differing class contexts (effectively giving them differing types). This next pass of analysis results in the graph in Figure 3.

3.2 Static Optimizations

The Concert system implements three interprocedural static optimizations [12, 34] to reduce object access overhead and enlarge thread granularity: object inlining, method inlining and access region expansion.

First, we apply *object inlining* to inline allocate object within other objects. Inline allocation lowers object access costs because the inlined objects' consistency can be managed by the container object. For example, Figure 4 shows the conjugations of an array of complex numbers, where object inlining enables access control of individual complex number objects to be merged with the array container (so `conj` becomes a function on `a` with `i` as a parameter). Object inlining also reduces storage management overhead because fewer objects need to be allocated and improves cache locality. Our adaptive analysis framework for handles dataflow through object state in order to inline allocate child objects even for polymorphic containers. It also allows systematic transforming of classes and replacing uses and definitions of inlined objects with inlined fields.

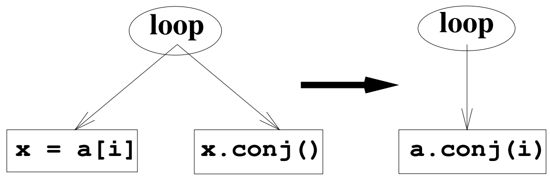

Figure 4. Object inlining transform

Secondly, because methods are typically small, we apply *method inlining* to eliminate method invocation overhead. To overcome polymorphism, we first clone methods based on calling environment to create opportunities for inlining. Because a method invocation can be inlined only if the target object is local and can be accessed, properties that are not always possible to determine at compile-time, we then speculatively inline by testing the required properties at run time. Figure 5 shows speculative inlining on the previous example, where runtime guards create *access regions* in their true arm, where locality and access control properties of the target objects are guaranteed (the `access?` node in Figure 5).

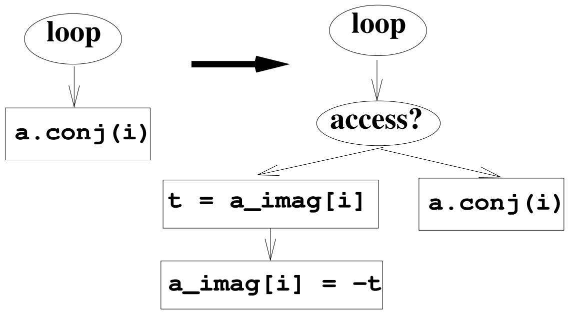

Figure 5. Method inlining transform

Lastly, because runtime checking can incur significant overhead if the access region is small or inside a loop, a third optimization, *access region expansion*, expands the dynamic extent of access regions to reduce overhead and additionally, creates larger basic blocks for scalar optimizations. Our optimizations both merge adjacent access regions and lift access regions above loops and conditionals, as shown in Figure 6, to create regions of optimized sequential code with the efficiency of a sequential uniprocessor implementation.

3.3 Locality Optimizations

Since global pointer-based data structures are fundamental for many dynamic (e.g. data-dependent) computations, Concert supports two locality optimizations [25, 43] to efficiently implement such structures on modern architectures with deep memory hierarchies, such as NUMA machines, whether cache-coherent or not. When static coarse-grained aliasing information is available, we apply *dynamic pointer alignment*, a generalization of static loop tiling and communication optimizations. When application data object access knowledge is available, we apply *view caching* to cache objects dynamically.

Dynamic Pointer Alignment exploits data reuse to reduce communication and tolerate remote access latency. Dynamic pointer alignment generalizes traditional loop-based strip-mining and tiling; it constructs logical iterations – actually light-weight threads – at compile time from loop bodies and function calls. At run time, the program concurrency structure allows these iterations to be reordered dynamically, guided by runtime data access information, to maximize data reuse and hide communication latency.

View caching [25] supports efficient runtime object caching in dynamic computations, relying on application knowledge of data access semantics to construct customized latency-tolerant coherence protocols that require reduced message traffic and synchronization. Application knowledge is used to infer information about the global state of objects and their copies, eliminating the need to acquire it at runtime. View caching decomposes coherence operations into three components — *access_grant*, *access_revoke*, and *data_transfer* — and builds protocols optimized for particular access patterns by putting together customized implementations of each component (selected from among a predefined set using application information).

3.4 Efficient Dynamic Multithreading

We exploit close coupling between the compiler and runtime systems to optimize logical threads in our non-binding concurrency model with respect to both sequential and parallel efficiency. Our hybrid stack-heap execution model [33, 26] provides a flexible runtime interface to the compiler, shown in Table 1, allowing 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. This allows the sequential portion of the program, where all accessed data is local, to execute with the efficiency of static procedure calls and the parallel portions to use efficient multithreading among heap-allocated threads. To separately optimize for sequential and parallel efficiency, the compiler generates two code versions for

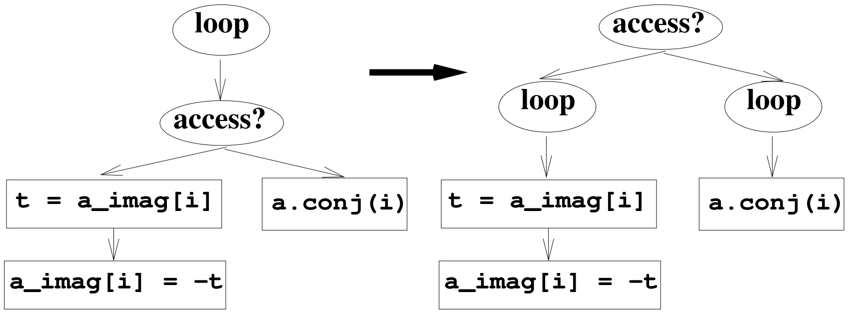

Figure 6. Region lifting transform

each function: a stack and a heap version. By analyzing the thread invocation structure, the compiler can choose the most efficient calling schema (among the stack versions), reducing the cost of a call without fallback to within a few instructions of a C function call.

VERSION	BASIC OPERATION
Heap	Most general schema, thread args and linkage through heap-allocated contexts
Stack	
Non-blocking	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

Table 1. Various thread interaction schemas in the hybrid stack-heap execution model.

3.5 Fast Communication and Thread Scheduling

To support fine-grained, distributed programs efficiently, the Concert implementation is built atop Fast Messages (FM) [24], which utilizes novel implementation techniques such as receiver-initiated data transfer to support high-performance messaging in the face of irregular communication that is unsynchronized with ongoing computation (a consequence of our dynamic programming model). These low-overhead, robust communication primitives support fine-grained computations efficiently, affording the compiler the flexibility to generate fine-grained remote object accesses interleaved with computation.

In addition, recognizing that application load-balance and thread scheduling is sometimes best managed by the application, our runtime system provides hooks for user-defined schedulers. The interface allows any number of customized schedulers to be integrated with the default runtime FIFO scheduler, permitting the application architect to flexibly manage thread scheduling and load balancing as dictated by application requirements.

4 Performance

We describe in turn the sequential and parallel performance of the Concert system.

4.1 Sequential Performance

We consider two sets of benchmarks to evaluate the effectiveness of the Concert system at reducing sequential overheads arising from high-level programming features. All numbers were taken on a Sparc 20/61.

Figure 7 (left) shows the performance for the OOPACK kernels. These kernels, designed specifically to test a compiler's ability to eliminate object-oriented overhead, come in two versions: a straightforward procedural implementation and an OOP one. Our system eliminates the overhead of object-orientation (encapsulation, small functions, and implicit storage-management) using static analysis and transformations (Section 3.2) to deliver similar performance on the procedural and OOP kernels. In contrast, g++ does a poor job of eliminating the OOP overheads.

Figure 7 (right) compares the performance of Concert to g++ on four complete programs: Silo, from the repository at Colorado, and Richards, Projection and Chain programs from DeltaBlue. These programs utilize a variety of sequential high-level features whose overhead is eliminated by Concert static transformations; cloning, method inlining, and object inlining are the major contributors. These optimizations yield performance ranging from slightly better than g++ on Silo to several times faster on DeltaBlue.

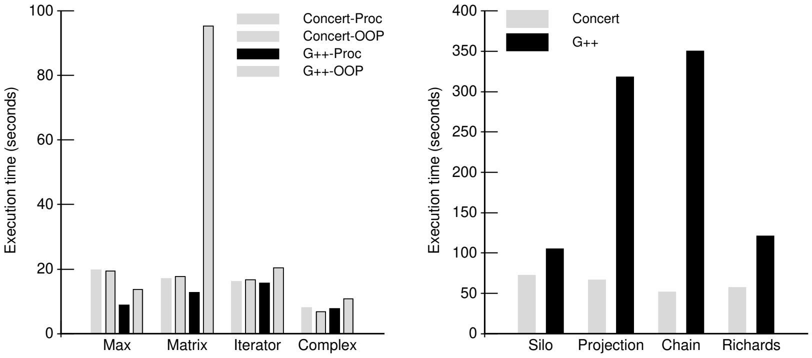

Figure 7. Performance of Concert and g++ on OOPACK kernels (left) and four benchmarks (right).

Table 2 shows, for each program, the important static optimizations contributing to the good sequential performance.

STATIC OPTIMIZATION	APPLICATION PROGRAMS				
	Oopack	**Silo**	**Projection**	**Chain**	**Richards**
Cloning	√	√	√	√	√
Object Inlining	√	√			√
Method Inlining	√	√	√	√	√

Table 2. Static optimizations contributing to good sequential performance.

4.2 Parallel Performance

We examine the performance of five large ICC++ application programs (shown in Figure 8(left)), spanning a range of computational domains, based on the Cray T3D implementation of the Illinois Concert System. High-level language features significantly simplify the program expression as compared to the original message-passing (**IC-Cedar**), and shared-memory (**Grobner, Radiosity, Barnes, FMM**) counterparts. The global object-space and implicit concurrency control eliminate the need to explicitly manage communication (for message-passing) and locking (for shared-memory). Non-binding concurrency and implicit task granularity help the expression of irregular task structure for all the programs. All programs achieve good sequential performance (within a factor of 2 of corresponding C programs), using access-region merging to eliminate overheads of the global object-space, and method and object inlining to eliminate overheads of object-orientation. Hybrid stack-heap execution reduces overheads of the implicit, non-binding concurrency specification.

Figure 8 shows the speedup of the applications with respect to the non-overhead portion of the single node execution time. The applications exhibit good speedups ranging from 8.5 on 16 nodes for **Grobner** to 54.8 on 64 nodes for the force phase of **FMM**. These speedups compare favorably with the best speedups reported elsewhere for hand-optimized codes [4, 35, 36, 21, 37]. For example, the **Radiosity** speedup of 23 on 32 T3D processors compares well with the previously reported speedup of 26 on 32 processors of the DASH machine [36], despite hardware support for cache-coherent shared memory and an order of magnitude faster communication (in terms of processor clocks) in the DASH which better facilitates scalable performance.

The good parallel performance is the aggregate effect of several optimizations which eliminate the overheads of Concert's high-level features. All of the static optimizations contribute significantly to achieve good sequential node performance for all five programs. In addition, Table 3 lists, for each runtime optimization, high-level feature(s) whose overhead it reduces, and whether or not the optimization was important for a specific program. Figure 9 shows the quantitative impact of each contributing optimizations for the **Radiosity** application: all the optimizations are essential (their absence results in a 35–80% performance drop), with different optimizations becoming more important at different processor configurations. For example, robust communication is important at small numbers of processors when communication traffic is high, and load-balancing is essential for large numbers of processors. Space limitations prevent us from a detailed analysis for the other applications; the reader is referred elsewhere [44, 26] for additional details.

5 Related Work

The Concert system is related to a wide variety of work on concurrent object-oriented languages that can be loosely classified as actor-based, task-parallel, and data-parallel.

PROGRAM	INPUT
Grobner Grobner basis	pavelle5 [4]
IC-Cedar Molecular dynamics	Myoglobin
Radiosity Hierarchical radiosity	Room [41]
Barnes Hierarchical N-body	16K bodies
FMM Hierarchical N-body	32K bodies

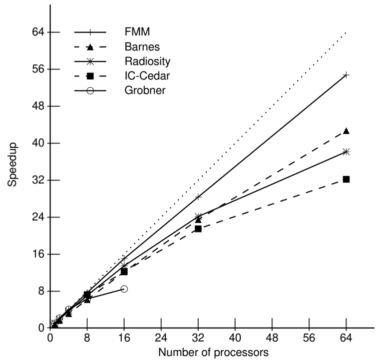

Figure 8. Speedup on the Cray T3D for five parallel ICC++ applications. Measurements for Barnes and FMM are only for the force phases. The speedup numbers are comparable to the best reported for low-level programming approaches.

OPTIMIZATIONS	HIGH-LEVEL FEATURES	APPLICATION PROGRAMS				
		Grobner	**IC-Cedar**	**Radiosity**	**Barnes**	**FMM**
Dynamic pointer alignment (§ 3.3)	global object-space, non-binding concurrency, implicit thread granularity		√		√	√
Object (View) caching (§ 3.3)	global object-space	√	√	√	√	√
Hybrid stack-heap execution (§ 3.4)	non-binding concurrency, implicit thread granularity	√	√	√	√	√
Robust messaging (§ 3.5)	global object-space	√		√	√	√
Thread scheduling for load balance (§ 3.5)	non-binding concurrency	√		√		

Table 3. Runtime optimizations contributing to good parallel performance.

Actor-based languages [1, 20, 42, 29] are most similar in terms of high-level programming support, but have focused less [39, 27] on efficient implementation. Task-parallel object-oriented languages, mostly based on C++ extensions [16, 23, 6], support irregular parallelism and some location independence, but require programmer management of concurrency, storage-management, and task granularity which limits scalability and portability. Data-parallel object-oriented languages, such as pC++ [28], provide little support for expressing task-level parallelism. HPC++ [2] is similar, expressing concurrency primarily as parallel operations across homogenous collections. ICC++ expresses data parallelism as task-level concurrency, providing greater programming power, but making efficient implementation significantly more challenging.

Concert differs from all the above systems in its focus on supporting high-level programming features with efficient implementation techniques. This focus can be found in the context of sequential object-oriented languages [11, 19, 5, 10], but our system additionally tackles the problems associated with concurrency, distribution and parallelism.

With respect to parallel systems in general, a wide variety of high-level approaches to portable programming are being actively pursued. Global address-space languages [14] minimally extend a low-level language with global pointers. While efficiently implementable, they require programmer control of distribution, concurrency, and task granularity. Data parallel approaches [40, 7, 18] express parallelism across arrays, collections, or program constructs such as loops in the context of a single control flow model. Such programs achieve efficiency by grouping and scheduling operations on colocated data elements. However, they cannot easily express task-level or irregular concurrency. Further, with the exception of Fortran 90 [7], data parallel languages provide no support for encapsulation and modularity.

6 Conclusions

We have described the Concert System, an optimizing implementation for a concurrent object-oriented programming model. We detailed the features of our language, ICC++, that supports fine-grained concurrency and concurrent abstractions. We also explained how our implementa-

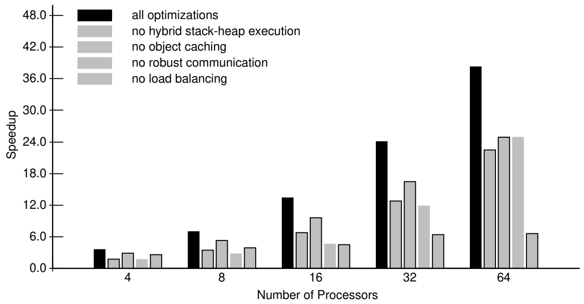

Figure 9. Quantitative impact of contributing runtime optimizations for the Radiosity application. Absence of an optimization results in a 35–80% performance drop.

tion uses a combination of compile-time static analysis and transformation, dynamic adaptation at runtime, and efficient runtime primitives to support the high-level language features without sacrificing on performance. We showed performance results showing that our approach achieves both high sequential and parallel performance.

Acknowledgements

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 grants MIP-92-23732, ONR grants N00014-92-J-1961 and N00014-93-1-1086 and NASA grant NAG 1-613. 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 is supported in part by an IBM Computer Sciences Cooperative Fellowship.

References

[1] P. America. POOL-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[2] P. Beckman, D. Gannon, and E. Johnson. Portable parallel programming in HPC++. Available online at `http://www.extreme.indiana.edu/hpc%2b%2b/docs/ppphpc++/icpp.ps`, 1996.

[3] B. Calder, D. Grunwald, and B. Zorn. Quantifying differences between C and C++ programs. Technical Report CU-CS-698-94, University of Colorado, Boulder, January 1994.

[4] 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.

[5] C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, Stanford, CA, March 1992.

[6] K. M. Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. In *Proceedings of the Fifth Workshop on Compilers and Languages for Parallel Computing*, New Haven, Connecticut, 1992. YALEU/DCS/RR-915, Springer-Verlag Lecture Notes in Computer Science, 1993.

[7] Chen and Cowie. Prototyping FORTRAN-90 compilers for massively parallel machines. In *Proceedings of SIGPLAN PLDI*, 1992.

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

[9] 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*, March 1996.

[10] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programmin g Language Design and Implementation*, pages 93–102, La Jolla, CA, June 1995.

[11] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Eleventh Symposium on Principles of Programming Languages*, pages 297–302. ACM, 1984.

[12] J. Dolby. Automatic inline allocation of objects. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.

[13] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[14] A. K. et al. Parallel programming in Split-C. In *Proceedings of Supercomputing*, pages 262–273, 1993.

[15] A. Goldberg and D. Robson. *Smalltalk-80: The language and its implementation*. Addison-Wesley, 1985.

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

[17] C. S. A. Group. The ICC++ reference manual. Concurrent Systems Architecture Group Memo. Available from http://www-csag.cs.uiuc.edu/, May 1996.

[18] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for FORTRAN D on MIMD distributed-memory machines. *Communications of the ACM*, August 1992.

[19] U. Hölzle. *Adaptive Optimization for* SELF*: Reconciling High Performance with Exporatory Programming*. PhD thesis, Stanford University, Stanford, CA, August 1994.

[20] 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*, pages 158–165, St. Charles, IL, August 1992.

[21] Y.-S. Hwang, R. Das, J. Saltz, B. Brooks, and M. Hodoseek. Parallelizing molecular dynamics programs for distributed memory machines. *IEEE Computational Science and Engineering*, pages 18–29, Summer 1995.

[22] G. L. S. Jr. *Common LISP: The Language*. Digital Press, second edition, 1990.

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

[24] 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. Available from `http://www-csag.cs.uiuc.edu/papers/cm5-t3d-messaging.ps`.

[25] V. Karamcheti and A. A. Chien. View caching: Efficient software shared memory for dynamic computations. In *Proceedings of the International Parallel Processing Symposium*, 1997.

[26] V. Karamcheti, J. Plevyak, and A. A. Chien. Runtime mechanisms for efficient dynamic multithreading. *Journal of Parallel and Distributed Computing*, 37:21–40, 1996.

[27] W. Y. Kim and G. Agha. Efficient support for location transparency in concurrent object-oriented programming languages. In *Proceedings of the Supercomputing '95 Conference*, San Diego, CA, December 1995.

[28] J. Lee and D. Gannon. Object oriented parallel programming. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, 1991.

[29] S. Murer, J. A. Feldman, C.-C. Lim, and M.-M. Seidel. pSather: Layered extensions to an object-oriented language for efficient parallel computation. Technical Report TR-93-028, International Computer Science Institute, Berkeley, CA, June 1993 Nov. 1993.

[30] ORB 2.0 RFP Submission. Technical Report Document 94.9.41, The Object Management Group, 1994.

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

[32] 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*, pages 324–340, 1994.

[33] 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.

[34] 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*, pages 311–321, January 1995.

[35] 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.

[36] J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7):45–56, July 1994.

[37] J. P. Singh, C. Holt, J. L. Hennessy, and A. Gupta. A parallel adaptive fast multipole method. In *Proceedings of Supercomputing Conference*, pages 54–65, 1993.

[38] Sun Microsystems Computer Corporation. *The Java Language Specification*, March 1995. Available at http://java.sun.com/1.0alpha2/doc/java-whitepaper.ps.

[39] K. Taura, S. Matsuoka, and A. Yonezawa. StackThreads: An abstract machine for scheduling fine-grain threads on stock CPUs. In *Joint Symposium on Parallel Processing*, 1994.

[40] Thinking Machines Corporation. *Getting Started in CM Fortran*, 1990.

[41] 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*, pages 24–36, 1995.

[42] 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 A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 55–89. MIT Press, 1987.

[43] X. Zhang and A. A. Chien. Dynamic pointer alignment: Tiling and communication optimizations for parallel pointer-based computations. Submitted for publication, 1996.

[44] X. Zhang, V. Karamcheti, T. Ng, and A. Chien. Optimizing COOP languages: Study of a protein dynamics program. In *IPPS'96*, 1996.