Efficient Implementation of High-Level Languages on User-Level Communication Architectures

by

Wilson C. Hsieh, Kirk L. Johnson,
M. Frans Kaashoek, Deborah A. Wallach, and William E. Weihl

May 1994

Abstract

User-level communication architectures — parallel architectures that give user code direct but protected access to the network — provide communication performance that is an order of magnitude higher than previous-generation message-passing architectures. Unfortunately, in order to take advantage of this level of performance, programmers must concern themselves with low-level issues that are often hardware-dependent (*e.g.*, what primitives to use for large and small data transfers, and whether to use interrupts or polling). As a result, programs are difficult to design, implement, maintain, and port. New compiler and runtime system mechanisms are needed to allow programs written in high-level languages — languages where the programmer does not orchestrate communication — to achieve the full potential of user-level communication architectures.

We propose a software architecture (compiler and runtime system) for implementing high-level languages with dynamic parallelism on user-level communication architectures. The compiler uses a simple runtime interface, and a new strategy called *optimistic active messages* to eliminate overhead due to context switching and thread creation. The runtime supports user-level message handlers and multithreading. We developed an implementation of the runtime for the CM-5; microbenchmarks demonstrate that our runtime has excellent base performance. We compare our compilation strategy and runtime with a portable runtime that uses traditional network interfaces. On our system, the microbenchmarks perform up to 30 times better; three hand-compiled applications run 10% to 50% faster. We also compare our approach on these applications with hand-crafted C programs that use active messages; the microbenchmarks perform within 25% of C, and almost all of the applications perform within a factor of two of C.

This work was supported in part by the Advanced Research Projects Agency under contracts N00014-91-J-1698 and DABT63-93-C-008, by the National Science Foundation under contract MIP-9012773, by a NSF Presidential Young Investigator Award, by grants from IBM and AT&T, by an equipment grant from DEC, by Project Scout under ARPA contract MDA972-92-J-1032, by a fellowship from the Computer Measurement Group, and by an Office of Naval Research Graduate Fellowship.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

1 Introduction

User-level communication architectures — parallel architectures that give user code direct, protected access to the network — are becoming common: direct access reduces the cost of communication, protected access enables more general use. Examples include the CM-5 [26], Hamlyn [39], the M-Machine [17], Nectar [3], Shrimp [11], and Typhoon [29]. Once user code has direct, protected access to the network, communication performance improves by an order of magnitude, because the costs associated with the operating system are eliminated. Reducing communication overhead enables the execution of finer-grained computations, which in turn allows problems to scale to greater numbers of processors.

Unfortunately, when communication is exposed to the user level, programming effort increases. In addition to issues regarding data and computation distribution, programmers have a new set of concerns to manage. They must be aware of many tradeoffs involved in using the network (*e.g.*, interrupts vs. polling, large vs. small data transfers, synchronous vs. asynchronous communication), worry about message buffering, and the potential for deadlocking the network [36]. In addition, much of this detail is hardware-dependent. Consequently, programs that exploit user-level communication are often hard to design, implement, debug, maintain, and port.

One approach to simplifying the programmer's task is to have the compiler and the programming language hide the complexity of low-level communication details. A great deal of work in this direction has been done in distributed systems (*e.g.*, Argus [27], Midway [8], Munin [7], and Orca [5]). However, these systems were not built for user-level communication; they use traditional RPC packages [10] to access the network. Even with ATM networks connecting DECstations, the latency for an RPC is approximately 170 microseconds [35]. As a result, the grain sizes of parallel computations are necessarily coarse.

Communication overhead for user-level communication architectures, in contrast, is an order of magnitude lower. For example, the cost of sending an active message on the CM-5 is a few microseconds [37]. As a result, communication overhead incurred at user level is non-negligible; in fact, it dominates the cost of communication. A different approach is necessary to retain the benefits of high-level programming while still taking advantage of the available performance.

In this paper we propose a software architecture that combines ease of programming with high-performance user-level communication architectures. Our approach is to compile directly to a simple runtime interface that provides active messages and multithreading. We present a novel runtime mechanism, called *optimistic active messages*, that allows arbitrary user code to execute as a message handler. We need optimistic active messages because the interaction between active messages and multithreading is deadlock-prone; the use of optimistic active messages avoids deadlock, as we explain in Section 2.

Our approach allows programs to be written in a high-level language, insulating the programmer from the complex low-level details of communication while providing communication performance close to that obtained by hand-coded programs. A possible disadvantage of our approach is that the size of the object code increases, because we depend on the compiler to generate code to handle special-purpose communications protocols for user code.

The primary contribution of this paper is a new software architecture that exploits the capabilities of user-level communication architectures: it relies on the compiler to hide communications complexity from the programmer and provides high performance by utilizing specialized communications code. The central feature of this software architecture is a new runtime mechanism, optimistic active messages, that reduces the thread management overhead of traditional approaches; it is a generalization of active messages. We have built a prototype of this software architecture on the CM-5; the runtime is complete, and the compiler generates optimistic active message code for simple programs. We used this prototype to conduct experiments with several microbenchmarks and three applications.

The remainder of the paper is structured as follows. Section 2 describes our software architecture and optimistic active messages in more detail. Section 3 describes our prototype implementation on the CM-5. Section 4 describes the experiments that we performed to measure our system. Finally, Section 5 discusses related work, and Section 6 summarizes our results.

2 Software architecture

By compiling high-level parallel languages to low-level communication primitives, we can achieve both high performance and ease of programming. Examples of high-level parallel languages include ABCL/f [34], Concert [24], HPF [20], Jade [30], Orca [5], and Prelude [38], just to name a few [6]. In all of these languages, the emphasis is on expressive power and ease of programming. The compiler must take full advantage of user-level communication in order for programs written in these languages to perform well.

Existing implementations of high-level parallel languages insulate the programmer from the details of the underlying communication system, but at a significant cost in performance. In general, executing a message handler involves many steps: marshaling the arguments, sending the message, receiving the message, creating or finding a thread to process the message, context switching to the new thread, unmarshaling the arguments, processing the message, marshaling the results, sending the reply, receiving the reply, context switching to the sending thread to process the reply, and unmarshaling the results. All but the actual

Operation	Time (μs)
Sender outgoing	
RTS layer	27
Marshal	38
Protocol layer	20
Send Data	101
Receiver incoming	
Wakeup thread	23
Protocol layer	20
Unmarshal	35
RTS layer	28
Receiver outgoing	
RTS layer	29
Marshal	17
Protocol layer	8
Send data	75
Sender incoming	
Wakeup thread	25
Protocol layer	86
Unmarshal	12
RTS layer	36

Table 1. This table shows the times associated with the various runtime operations involved in executing a null application-level RPC in Orca; these numbers include measurement overhead.

processing of the message are pure overhead. User-level communication architectures greatly reduce the costs of sending and receiving messages by bypassing the operating system. The major remaining sources of overhead are in thread management (thread creation, scheduling, and context switching) and in marshaling and unmarshaling of data.

Table 1 contains typical method invocation overheads that we measured on the Thinking Machines CM-5 [26]; the numbers are a breakdown of the time that the Orca system [5] spends during a null method invocation on a remote object.[1] Further optimization of the runtime system is possible, but keeping a high-level interface (such as RPC) to the communication system still limits performance; a different approach is necessary to achieve higher performance.

One approach is to use runtime libraries such as active messages, which execute user code directly in message handlers, instead of creating a thread to do so. Writing programs that use active messages, however, is difficult: the addition of multithreading introduces deadlock issues. In this section we describe a new

[1]This runtime is an optimized version of the Panda runtime system for Orca [9] that we ported to the CM-5.

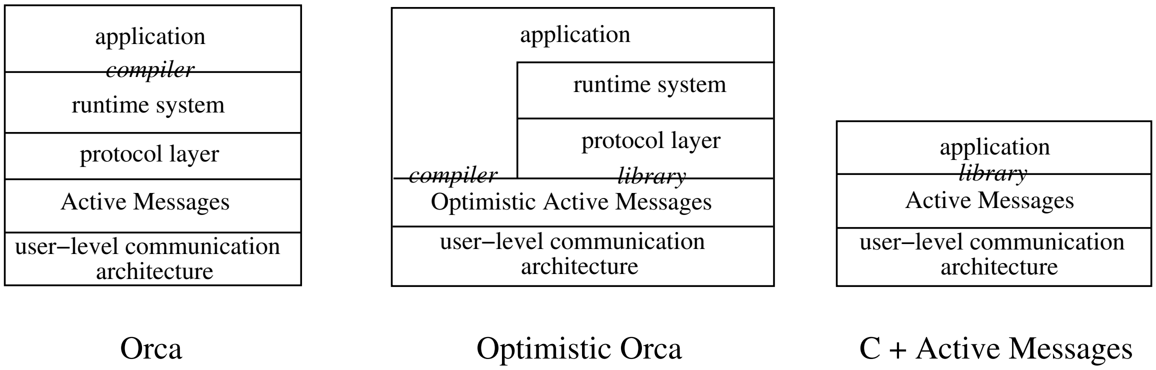

Figure 1. Layers in different message-passing systems. Some of the layer boundaries are library interfaces; others are calls generated by the compiler. The protocol layers implement the RPC and group communication protocols.

runtime mechanism, called *optimistic active messages* (OAM), that allows arbitrary user code to execute in a message handler, thus avoiding thread management overhead and allowing multithreading without the potential for deadlock. We also describe compilation techniques for eliminating marshaling overhead. We built a prototype system that uses this novel runtime mechanism for Orca; we call this system *Optimistic Orca*.

Our compilation strategy generates code for a simple communication and thread management interface provided by the runtime system. The interface supports active-message-style communication as well as multithreading (using either lightweight threads as in Pthreads [28] or even lighter-weight threads as in TAM [15]). Our techniques can be applied to any system that provides similar communication and thread management support. The differences between our approach and the strategies of providing a general communication abstraction (as in previous Orca implementations) and of providing the programmer with a low-level communication library (*e.g.*, C and active messages) are illustrated in Figure 1. We experimented with adding optimistic active messages directly to the Orca runtime; this reduced the time for null RPC by a factor of two, which is small compared to the gains of compiling directly to optimistic active messages.

2.1 Optimistic active messages

An *optimistic active message* runs arbitrary user code in a message handler. Optimistic active messages can be used for any language or system with locks and multithreading; however, to be concrete let us consider the problem of compiling programs in languages such as Amber [12], Concurrent Aggregates [14], Orca [5], or Prelude [38]. In such languages, objects and threads are created on processors; communication occurs when

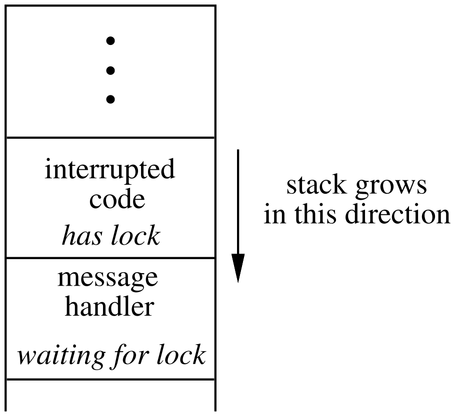

Figure 2. Scenario illustrating the potential for message handler deadlock.

a thread calls a method on a remote object. To eliminate thread management overhead in communication, we would like to run all remote method invocations in message handlers rather than dispatch to separate threads. However, there are several obstacles that make this difficult.

The most serious problem is deadlock. If a method running in a message handler blocks (*e.g.*, waiting to acquire a lock), deadlock could result (see Figure 2). Thus, user code that might block cannot be put directly in a message handler; either it must be analyzed to determine that deadlock cannot occur, or it must be compiled so that corrective action is taken if it blocks. In this paper we explore the latter approach.

As discussed by von Eicken et al. [37], it is also important for message handlers to be short. While a message handler is running on a processor, new messages that arrive at the processor cannot be processed. In high-performance multiprocessor interconnects such as on the CM-5, these messages will be blocked inside the network interface, and as more messages arrive, will back up into the network. Significant congestion could result if a message handler runs for too long and prevents the processor from removing other messages from the network.

Our solution to these problems is to be optimistic. We compile methods with the assumption that they usually run without blocking and complete quickly enough to avoid causing network congestion. When a method might block or runs for a long time, we include checks in the compiled code that detect these situations and revert to a slower, more general technique for processing the method (*e.g.*, creating a separate thread).[2] This approach eliminates thread management overhead for methods that run to completion within

[2]The definition of "too long" is architecture-dependent, but the compiler can account for these effects. It is also application-dependent: if the application sends messages relatively infrequently, its message handlers can run for longer without causing serious congestion. Accounting for these effects requires sophisticated compiler analysis.

```
acquire local lock on invoked object
    execute method body
release lock on invoked object
```

(a) Method code

```
in message handler:
    if lock is free, execute method body
        if code runs for too long, abort
    if lock is not available, abort
```

(b) Generated code

Figure 3. Locking Example

a message handler, and will improve performance relative to existing RPC systems as long as most remote method invocations do not block or run for too long.

2.1.1 Detecting the need for an abort

When a method cannot complete within a message handler, we say that the optimistic execution as an active message must *abort*. In our software architecture, there are three reasons why an OAM may have to abort: it needs to wait for a lock, it needs to send a message but the network is busy (*i.e.*, the network interface buffers are full), or it needs to run for a long time. Each of these conditions is straightforward to detect.

For example, in many object-based languages, such as Orca, Prelude, and Concurrent Aggregates, a method invocation on an object implicitly acquires a lock on the object before executing the body of the method; this guarantees method atomicity, which simplifies programming. This approach is illustrated by the pseudocode in Figure 3(a). A compiler that compiles to simple active messages must make the conservative assumption that the method could block on the initial lock acquisition, even if runtime conditions are such that the lock is usually free. This would mean that the code generated for the method must start a new thread to execute the body of the handler.

A compiler that generates code as optimistic active messages, on the other hand, generates message handler code analogous to the pseudocode in Figure 3(b). As long as the lock is usually free when a method invocation is received, and as long as the code does not take a long time to execute, the approach illustrated in the right half of Figure 3 will yield significantly better performance than previous approaches.

If a method also acquires other locks in addition to the lock acquired at the beginning, then the compiler can hoist the lock acquisitions to the top of the message handler. Although it is usually more efficient to hold locks for shorter periods of time, in this situation it is more efficient to check if all of the locks are free at the start of the OAM. If the locks are all free, then the atomicity of the message handler guarantees mutual exclusion; if one of them is not, a thread must be started.

Detecting that the network is busy is straightforward on many systems. For example, on the CM-5, the processor can inject data into the network, and then test status bits to determine whether the injection succeeded or failed. Existing communication libraries on the CM-5 spin when sending a message: they attempt to inject the message, and if it fails, they loop around and try again. (In some cases, they remove any pending data from the network before trying again.) In an optimistic active message handler, an attempt to send a message should check for success, and abort the handler if the send failed. It could also try more than once, up to some small limit; by analogy with "competitive" algorithms in other domains [31], it could retry until the time spent trying to send equals the overhead of aborting the message handler.

The compiler must insert code for an abort if a method runs for too long. The compiler can count the number of instructions in the method, and can insert abort code when the execution is too long. If the method contains procedure calls, either interprocedural analysis is necessary, or aborts can always occur at procedure call boundaries. If the method contains loops with non-constant bounds, the compiler can strip-mine them to ensure that the message handler does not exceed the desired time limit.

2.1.2 Aborting an OAM

There are three ways in which an OAM can be aborted:

1. Abort can create a continuation for the OAM; that is, execution of the method continues in a new thread. Effectively, the first part of the method runs in the message handler, and the remainder runs in a thread. The compiler can generate a procedure for the continuation; the aborting OAM starts a thread to run the continuation procedure. This method avoids redoing the work performed in the message handler, and can be viewed as the lazy creation of a thread.

2. The abort operation can undo the execution of the OAM (the OAM is treated as a transaction that must be undone, for which the compiler must generate code), and then start a thread that runs the entire method. This might be the correct choice if the OAM needs to recompute its state when it executes again: for example, the decision to block may depend on some state that may change. If the OAM does not modify any object state, then undo is particularly easy; otherwise, the compiler must restore the modified values.

3. The abort operation can undo the execution of the OAM, and send a negative acknowledgment back to the sender, who is then responsible for resending the OAM after some backoff period. This is analogous to test-and-set with exponential backoff, whereas the previous two possibilities are analogous to queue-based locking strategies. Although this possibility consumes more network bandwidth than the others and provides no fairness guarantees, it has the advantage that it relieves processor demand at the receiver. If the thread management overhead for the previous two alternatives

is high, forcing the sender to retry could be more efficient. Note that the backoff need not be coded by the programmer: the stub generator can put the backoff code in the stub for the method.

The correct choice depends upon the application, the language semantics, and in some cases the architecture. We experimented with the third choice, but found the second choice more efficient for our language model in our prototype system.

2.2 Reducing marshaling overhead

It is possible for the compiler to exploit the semantics of a program to further reduce the overhead of communication and synchronization. For example, the compiler can use its knowledge of the types of method arguments to reduce marshaling costs. Simple types, such as integers and floats, need not be marshaled. Because marshaling is an expensive component of message-sending overhead, this optimization can substantially reduce the cost of message-sending.

2.3 Discussion

Our software architecture for user-level communication simplifies parallel programming: the compiler and language hide the details of managing user-level communication. By hiding this complexity, our architecture enables programmers to write cleaner, more portable code. At the same time, we achieve high performance: the compiler generates code to use user-level communication primitives directly and uses optimistic active messages to avoid context switching and thread creation.

One disadvantage of our architecture is that the complexity of the compiler increases. However, since the back end of the compiler only needs to be written once per machine, we feel that this is a good tradeoff. A more significant disadvantage is that object code size increases, since the compiler must generate additional code to handle aborting OAMs. One way to reduce the code expansion might be to generate OAMs based on profiling information: measure the program to determine which methods are contributing the most overhead to the execution of the program, and compile only those methods as optimistic active messages. Another is to have the programmer use annotations to indicate important methods.

The use of optimistic active messages together with the elimination of most marshaling code reduces the overhead of communication significantly, but some overhead still remains. For example, active messages require either polling or interrupts, both of which involve significant cost. Pointer swizzling is another major cost for many systems. In addition, maintaining consistency among replicas is expensive. It should be possible to reduce these overheads through the use of compile-time and runtime analysis. Felten explored this issue for message buffering [19]; he terms this form of optimization protocol compilation. The Munin system [7] uses annotations to choose the appropriate consistency model; it may be possible to use compile-time or runtime analysis to automate this choice.

3 Implementation

In order to evaluate our approach, we have implemented a prototype of our software architecture on the CM-5. The CM-5 provides efficient user-level communication: user processes have direct access to the network interface. Protection between users is implemented by using strict gang scheduling and by context switching the state of the network along with that of the processors: in-transit packets are swapped out to the processors when a context switch occurs. In the remainder of this section we describe the Orca system. We obtained the sources for the Orca compiler, runtime system, and several applications from the Vrije Universiteit; we ported it to the CM-5 and used it as our base prototype for exploring optimistic active messages.

3.1 Orca

We use Orca because it is based on a combined compiler/runtime system approach [4], which makes it easy to move functionality into the compiler, and because it has been successfully ported to a number of other platforms. In the Orca system, threads share data through shared objects, which are instances of abstract data types. Although the object model of Orca is more restrictive than some others, it nonetheless provides valuable insight into the benefits of our approach. We indicate when we are taking advantage of an Orca-specific property.

An object that we use in our experiments is depicted in Figure 4. The object is an integer object with three operations: *inc*, *value*, and *AwaitValue*. The first two operations never block the calling thread, while the last one blocks the calling thread if v does not equal x.

The Orca system performs runtime migration and replication of objects based on information that the compiler deduces. The Orca compiler classifies operations as either *read* or *write*, where read operations do not modify an object (*e.g.*, *value* and *AwaitValue*) and write operations potentially modify an object (*e.g.*, *inc*). In addition, the compiler computes access patterns for each Orca thread that describe its sequence of read/write operations. Whenever a new thread is forked on an object, Orca decides whether to migrate or replicate the object to reduce communication latency: objects with a high read/write ratio are replicated; other objects are stored on the processor that is expected to perform the most write operations. Remote

```
OBJECT IMPLEMENTATION IntObject;
    x: integer;

    OPERATION value(): integer;
    BEGIN RETURN x; END;

    OPERATION inc();
    BEGIN x +:= 1; END;

    OPERATION AwaitValue(v: integer);
    BEGIN GUARD x = v DO OD; END;

BEGIN x := 0; END;
```

Figure 4. An Orca integer object that supports three operations.

objects are accessed through RPC, and replicated objects are updated using group communication.

3.2 Optimistic Orca

We have implemented Optimistic Orca, a prototype of our system, on the CM-5. To explore our ideas, we began by hand-compiling Orca methods to optimistic active messages for the applications that we describe in Section 4. We have since altered the compiler so that it can automatically generate optimistic active messages for certain simple abstract data types. Extending the compiler so that it can handle arbitrary data types should be straightforward.

In our prototype the programmer annotates objects to indicate whether they should be compiled using optimistic active messages. Objects that do not use optimistic active messages are compiled in the same manner as in the original Orca system.

The compiler classifies the methods on an optimistic object (one compiled using optimistic active messages) as non-blocking or blocking. Non-blocking operations are compiled to active messages (our prototype assumes that the methods are sufficiently short), which avoids both thread management and marshaling costs. The cost of executing a method in this case is slightly greater than the corresponding cost in C, due to the overhead of the object model: global object identifiers must be swizzled.

In Orca, a blocking method can only block immediately after the evaluation of a guard (if the guard is false). The compiler places the evaluation of the guard in an optimistic active message. If the guard is true, the method is executed within the OAM (again, we assume that the method is sufficiently short). If the guard fails, the OAM aborts: the method's identifier and arguments are stored on a queue at the object.

System	RPC data size				
	4 bytes	**20 bytes**	**64 bytes**	**1024 bytes**	**4096 bytes**
Orca inc	525	680	725	1150	2340
Optimistic Orca inc	16.4	38	39	149	495
C+AM inc	12.7	30	29	137	495

Table 2. Comparison of Orca, Optimistic Orca, and C+AM: time (in μs) to perform a short, non-blocking operation, with varying argument sizes. (The fact that the 20-byte RPC is slightly slower than the 64-byte RPC for C+AM is due to anomalous behavior of *scopy*.)

Whenever a later method modifies the object, the later method reevaluates the guards of all of the methods on the queue; any method whose guard is true is executed at that time.

4 Experiments

In this section we evaluate the performance of our prototype. We compare our prototype system with applications written in both the original Orca system for the CM-5, as well as hand-coded C with active messages (henceforth referred to as C+AM). All three systems use Version 3.1 of the CMMD library [36], which implements active messages, running under Patch 1 of CMOST Version 7.2. Both Orca systems implement RPC and group communication using active messages for short messages; both protocols use the CMMD *scopy* primitive for bulk data transfer.[3] We implemented multithreading by porting Pthreads [28]. To allow a direct comparison to C+AM, we compiled Orca applications with array bounds checking turned off.

4.1 Microbenchmarks

To measure the benefits of Optimistic Orca, we developed a number of microbenchmarks to measure specific performance aspects of the Orca systems. These microbenchmarks study the performance of blocking and non-blocking operations on unreplicated objects.

The first microbenchmark, *inc*, measures the time to perform an increment operation on the integer object discussed previously; arguments of various sizes (which are ignored) are sent to measure the effect of argument size on performance. *Inc* is always non-blocking. Table 2 compares the performance of Orca, Optimistic Orca, and C+AM. The first column of the table, representing a four-byte argument to the RPC, shows the performance where active messages can be used to communicate the data instead of resorting

[3] Group communication is implemented using an active message broadcast tree and a sequencer.

to the bulk data transfer mechanism. Both the C+AM and Optimistic Orca versions take advantage of this optimization, and also avoid marshaling and thread management overhead; as a result, they perform 30 times better than Orca. The other columns show the performance when a bulk data transfer is required. This data transfer has a large constant overhead, due to the need for the invoker and invokee to negotiate the transfer. For these cases, the Optimistic Orca version is still 17 times faster than Orca for small arguments, and four times faster for larger ones, when the data transfer costs begin to dominate.

The second microbenchmark, *pingpong*, measures the number of blocking operations per second that can be performed on a remote object. Two threads cooperate in counting: each thread executes the operations *AwaitValue* and *inc* in a loop — each waits for the other to increment the integer, and then increments the integer itself. Because we perform this experiment with two processors, half of the requests are remote, and the other half are local. The Orca system can perform 1.9 counts per millisecond (each count consists of an *AwaitValue* operation followed by an *inc* operation); the Optimistic Orca system can perform 34.5 counts, nearly a factor of 20 faster. This improvement is due to two reasons: all the remote *inc* invocations are executed as active messages, and all the remote *AwaitValue* invocations are executed as optimistic active messages, which block and create lightweight continuations.

In contrast, the Orca version awakens a separate thread for both the remote *inc* and *AwaitValue* operations. Furthermore, invocations pass through several communication abstraction layers, and incur the concomitant cost. Finally, the Optimistic Orca system takes advantage of the fact that small amounts of data need to be transferred for each operation, so the slow, block transfer protocol can be bypassed in favor of a single active message.

4.2 Application performance

In this section we describe the performance improvements due to the use of optimistic active messages for a number of Orca applications of varying communication patterns and shared-data use. These are the same applications used by Bal and Kaashoek to measure the performance of the original Orca system [4]. Because they were originally developed for distributed systems, which have relatively high communication overhead, they are relatively coarse-grained. We ran the applications at finer grain sizes than they did in order to study how well our prototype could support lower grain size. Our measurements show that the use of optimistic active messages results in performance improvements for all of these applications.

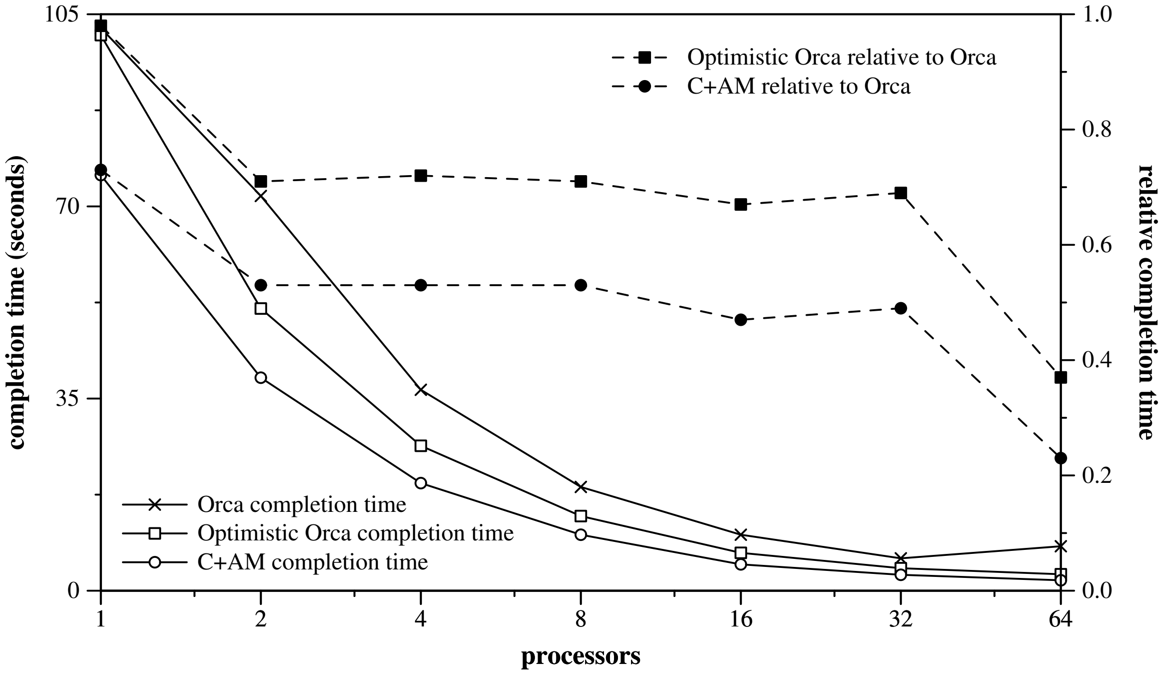

Figure 5. Performance of All-Pairs Shortest Paths using Orca, Optimistic Orca, and C+AM, problem size of 512 nodes. In addition to absolute completion time, the graph depicts completion time relative to Orca.

4.2.1 All-Pairs Shortest Paths

The first application is All-Pairs Shortest Paths (ASP), which finds the length of the shortest paths between every two nodes in a graph with N nodes. The distances between the nodes are stored in a matrix. Each processor contains a worker process that computes part of the result matrix. The parallel algorithm performs N iterations. Before each iteration, one of the workers sends a pivot row of the matrix to all the other workers. The pivot row contains N integers and is broadcast to all processors using group communication. We ran our experiments with 512 nodes.

A sequential C version of this program ran in 65.7 seconds on a 33 MHz CM-5 node. As Figure 5 illustrates, Optimistic Orca runs at worst 1.6 times slower than C+AM. It is not evident from the graph, but the gap slowly widens as the number of processors increases; as the amount of communication increases, the overhead that Optimistic Orca incurs is still significant with respect to C+AM. Some of this overhead is due to Orca's dynamic object model, which is absent from C+AM: in particular, the support for dynamic data structures in Orca results in extra copying of the rows that are broadcast. Given that Optimistic Orca hides the low-level communication details, we consider this to be excellent performance.

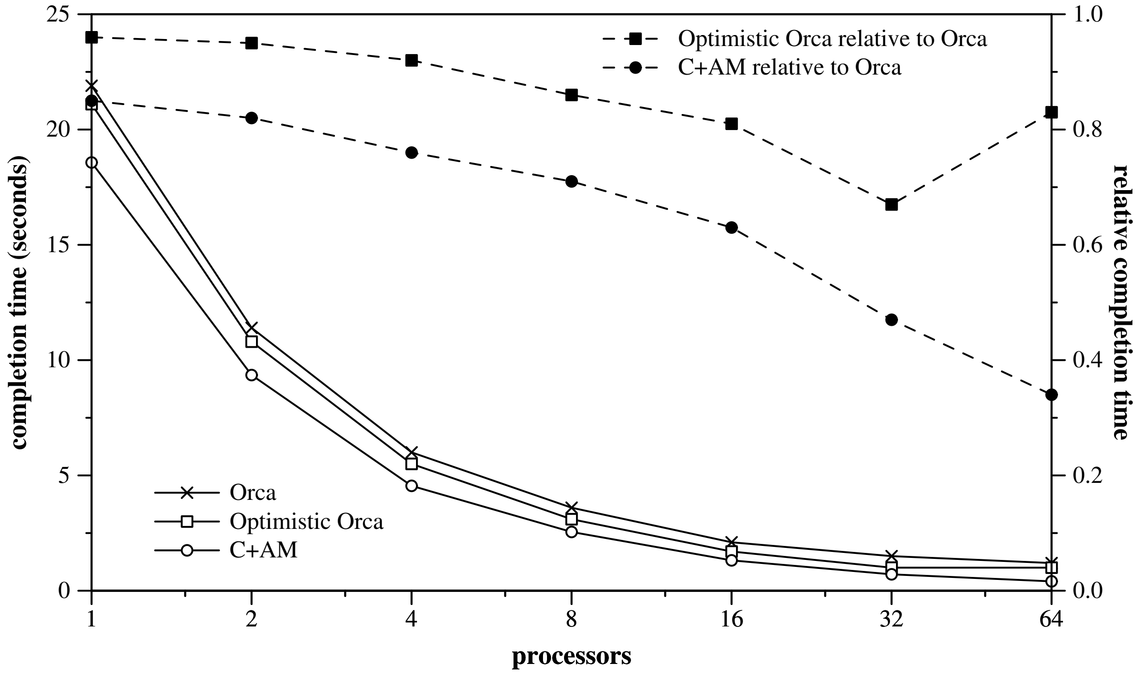

Figure 6. Performance of Successive Overrelaxation using Orca, Optimistic Orca, and C+AM, problem size of 482$\times$80. In addition to absolute completion time, the graph depicts completion time relative to Orca.

Optimistic Orca is approximately 33% faster than Orca. The relative completion times using both Optimistic Orca and C+AM drop significantly at 64 processors because the completion time for Orca actually increases from 32 to 64 processors. This is due to the combination of Orca's poor communication performance and its less efficient broadcast scheme.

4.2.2 Successive Overrelaxation

Successive Overrelaxation (SOR) is an iterative method for solving discrete Laplace equations on a grid. During each iteration, the algorithm updates all of the non-boundary points of the grid with the average value of its four neighbors. After each iteration, all workers determine if they have converged; when they have, the program terminates. For a more direct comparison, we use two special primitives of the CM-5 to implement the convergence test in all three implementations: the global OR set and get pair, which provides a split-phase barrier, and a global reduction to implement the vote.

We ran a problem size of 482 rows by 80 columns. The grid is partitioned along the row axis; neighboring threads exchange 80 double-precision floats every iteration. The sequential C version of this program runs in 19 seconds.[4] As shown in Figure 6, the Optimistic Orca version is consistently 10 to 30% faster than the Orca version. The performance of Optimistic Orca bottoms out at 32 nodes: increasing the number of processors further does not lower the completion time.[5] The hand-coded version exhibits the best performance (between 15 to 30% faster than Optimistic Orca for up to 32 processors) because when rows are exchanged they are sent directly to the appropriate locations in memory; in Orca, the programming model requires that each row is copied out of an intermediate data structure that is used for sharing.

4.2.3 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a search problem that finds the shortest Hamiltonian tour of a set of cities. The problem is solved in Orca using a master/slave style program based on a branch-and-bound algorithm. The master generates partial routes and stores them in a job queue. Each slave repeatedly takes a route from the queue and generates all possible full paths by using the "closest-city-next" heuristic. All slaves keep track of the shortest route that has been found, which is used to prune part of the search tree.

We normalized the comparison by presetting the shortest route bound to be equal to the length of the actual minimum route, which we computed in advance. This leads to a deterministic search, regardless of relative speeds of the slaves; if we did not normalize the problem, the actual order of the search would affect the overall timings.

We show the results for a 12-city problem, in which the master created 969 partial routes of length four. A sequential C version of the code solved this problem in 11.7 seconds. As can be seen in Figure 7, both the Orca version and the Optimistic Orca version scale well; Optimistic Orca is twice as fast as Orca. Optimistic Orca is approximately half as fast as C+AM; this is partially because the C+AM version has a more efficient job queue implementation.

To confirm that the lower communication overhead of Optimistic Orca allows the exploitation of finer-grained parallelism, we reduced the grain size in TSP by having the master generate partial routes of length five (6912 jobs, same sequential running time). For up to 16 processors with the finer-grained problem, Optimistic Orca performs between two and three times better than Orca; C+AM is only 10 to 30% faster than Optimistic Orca. At 32 processors, both the Orca versions are limited by the throughput of the job queue on the master.

---

[4] In the timings shown for all versions of our SOR applications, the innermost functions are not inlined, in order to provide consistent timings. Both C+AM and Orca programs seem to speed up by a factor of about 1.5 with the inlining.

[5] We do not fully understand the behavior of Optimistic Orca at 64 processors; we are currently investigating this behavior.

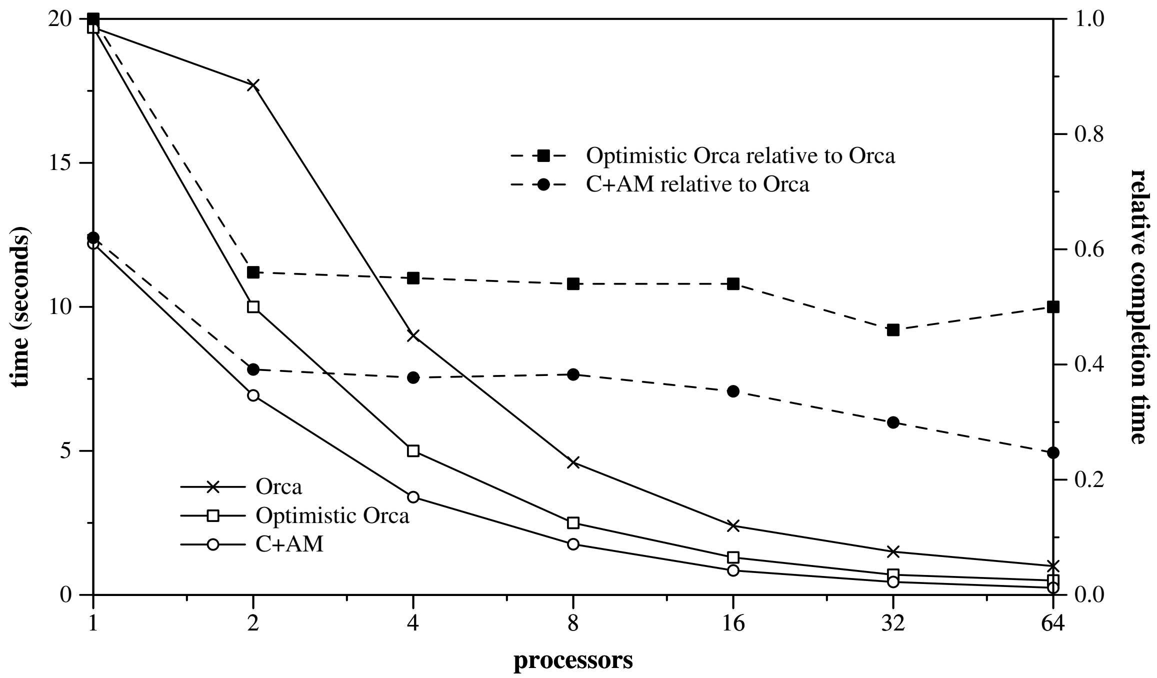

Figure 7. Performance of Traveling Salesman Problem using Orca, Optimistic Orca, and C+AM, problem size of 12 cities, where the master generates 969 partial routes of length four. In addition to absolute completion time, the graph depicts completion time relative to Orca.

4.2.4 Summary of experiments

The microbenchmarks show that Optimistic Orca's communication performance is very close to that of C+AM, and an order of magnitude better than that of Orca. In the applications, Optimistic Orca is twice as fast as Orca. Optimistic Orca is approximately half as fast as C+AM (except for the one data point for SOR at 64 processors); since the communication performance of both systems are similar, our performance is suffering in other areas. For example, in TSP the C+AM job queue implementation is more efficient; in SOR the Orca model requires an extra data structure through which shared rows are communicated.

5 Related work

As mentioned in the introduction, a variety of machines provide (or will provide) protected user-level communication: the CM-5 [26], Hamlyn [39], the M-Machine [17], Nectar [3], Shrimp [11], and Typhoon [29]. There have been several other proposals for protected user-level communication: Spector proposed a model for executing remote operations on a LAN [32]; more recently, Cheriton and Kutter have proposed using shared memory to perform user-level messaging [13]. However, none of the systems address the issue of programming the communication, which can be quite difficult; the increasing availability of user-level communication architectures will increase the demand for a high-level programming model that provides good performance.

Alewife [1], the J-Machine [16], and Tera [2] are examples of hardware-intensive approaches to fast communication and multithreading. Our approach is a software alternative that does not depend on the existence of a limited number of thread contexts; the work on active messages indicates that little hardware is needed to integrate computation and communication. Mul-T [25] and CST [22], two high-level languages that were implemented on Alewife and the J-Machine, respectively, hide communication details and provide good performance by exploiting the special-purpose hardware on those machines. Our approach does not depend on special hardware other than user-level communication.

Fortran D [21], Vienna Fortran [40], and HPF [20] are data-parallel variants of Fortran that hide the complexity of communication from the programmer: the compilers for these languages generate explicit send/receive code for communication. These languages are designed for data parallel programs, which are more amenable to static analysis, and they compile to a more limited form of communication. Our software architecture is designed for more general parallel languages, where the compiler cannot determine which methods are blocking.

Felten [19] explored the area of protocol compilation. His compiler analyzed communication patterns to minimize communication and synchronization overhead, in particular, message buffering. Felten's work differs from ours in that he compiled to a send/receive interface. Implementing Felten's techniques in our compiler should result in similar benefits in our system.

There has been a great deal of work in distributed systems in hiding the low-level details of communication from the programmer (*e.g.*, Amber [12], Argus [27], Midway [8], Munin [7], and Orca [5]). Our work is similar in spirit, but these other systems were designed for use on traditional distributed systems, which have communication costs that are much higher than those on user-level communication architectures.

In the Split-C compiler [37], remote memory operations are directly compiled into active messages; the compiler provides a primitive version of shared memory that provides fast remote references through active messages. Von Eicken *et al.* also developed active messages, upon which optimistic active messages is based.

There have been a number of high-level languages for parallel programming (*e.g.*, Jade [30], Concert [24], ABCL/f [34], Prelude [38], and many others). These systems could all make use of optimistic active messages. The Jade language is implemented on top of a portable runtime system called SAM. Jade is designed for coarse-grain applications, and the compiler does not compile directly to communications primitives, as we do.

In the Concert system, methods that are known to be non-blocking at compile-time are compiled to active messages. In addition, the Concert system uses compile-time and runtime techniques to select the cheapest alternatives for doing method dispatch. For blocking operations, Concert would benefit from using optimistic active messages.

The ABCL/f language compiles method invocations to a variant of optimistic active messages. When a message handler is invoked, it executes on the stack; if it blocks, its frame is then copied to the heap. This requires more runtime support than our system; our prototype compiler generates the necessary code for the continuation. ABCL/f uses this facility for local as well as remote invocation, which allows efficient fine-grained multithreading.

In the Prelude language certain methods are always known to be short: these are methods that get and set fields within objects. Gets and sets on non-atomic objects (where methods do not implicitly acquire a lock) are compiled directly to active-message-like code; those on atomic objects are compiled to a form of optimistic active messages.

Having the compiler generate continuations is similar to what was done in the Prelude compiler to support computation migration [23]. More broadly, our use of continuations is similar to the introduction of continuations into operating system structure [18]. Our use of optimistic execution is also similar to work done on interrupt handling [33].

6 Conclusion

We have described a software architecture for user-level communication architectures that is designed for the compilation of high-level languages. By making our source language high-level, we hide low-level communication issues from the user. We have described the structure of the runtime and the compiler; the compiler uses a strategy we call optimistic active messages to make more efficient use of message handlers.

We built a prototype implementation of the runtime on the CM-5; this implementation evolved from work on versions of the Orca system. We then hand-compiled three parallel applications in Orca (Traveling Salesman, Successive Overrelaxation, and All-Pairs Shortest Paths) for our runtime and modified the Orca compiler so that it could automatically generate optimistic active messages for certain simple abstract data types. We compared the performance of the same applications running on Orca, as well as with implementations written in C with active messages. The experiments demonstrate that our techniques lower the cost of using message passing in Orca by 25 to 100%; the resulting performance is within a factor of 2 of hand-crafted C code, with one exception.

Microbenchmarks demonstrate that Optimistic Orca's raw communication performance is up to 30 times better than Orca's, and is only 30% slower than C with active messages. This implies that the difference in application performance between Optimistic Orca and C is due to artifacts of Orca, both in the implementation and language. We expect that a new language and system based on our architecture will outperform Optimistic Orca, and will achieve application performance close to that C.

Acknowledgments

We thank the Orca group at Vrije Universiteit for developing the system with which we started, and in particular Raoul Bhoedjang, who did the first Panda port to the CM-5. Thanks are also due to Eric Brewer, Dawson Engler, and Kevin Lew for providing comments on early drafts.

References

[1] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. "The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor". In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. Extended version is MIT/LCS/TM-454.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. "The Tera Computer System". In *Proceedings of the International Conference on Supercomputing*, pages 272–277, Amsterdam, The Netherlands, June 1990.

[3] E.A. Arnould, F.J. Bitz, E.C. Cooper, H.T. Kung, R.D. Sansom, and P.A. Steenkiste. "The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers". In *Proceedings of the 3rd Conference on Architectural Support for Programming Languages and Systems*, pages 205–216, Boston, MA, April 1989.

[4] H.E. Bal and M.F. Kaashoek. "Object Distribution in Orca using Compile-Time and Run-Time Techniques". In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '93*, pages 162–177, September 26–October 1, 1993.

[5] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. "Orca: A Language for Parallel Programming of Distributed Systems". *IEEE Transactions on Software Engineering*, 18(3), March 1992.

[6] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. "Programming Languages for Distributed Computing Systems". *ACM Computing Surveys*, 21(3):261–322, September 1989.

[7] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence". In *Proceedings of the 2nd Symposium on Principles and Practice of Parallel Programming*, March 1990.

[8] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. "The Midway Distributed Shared Memory System". In *Proceedings of COMPCON Spring 1993*, pages 528–537, San Francisco, CA, February 22–26, 1993.

[9] R. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H. Bal, and F. Kaashoek. "Panda: A Portable Platform to Support Parallel Programming Languages". In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 213–216, San Diego, CA, September 22–23, 1993.

[10] A.D. Birrell and B.J. Nelson. "Implementing Remote Procedure Calls". *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[11] M.A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E.W. Felten, and J. Sandberg. "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer". In *Proceedings of the 21st International Symposium on Computer Architecture*, Chicago, IL, April 18–21, 1994. To appear.

[12] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. "The Amber System: Parallel Programming on a Network of Multiprocessors". Technical Report 89-04-01, Univ. of Washington Dept. of Computer Science, April 1989.

[13] D.R. Cheriton and R.A. Kutter. "Optimizing Memory-Based Messaging for Scalable Shared Memory Multiprocessor Architectures". Publication status unknown.

[14] A.A. Chien. "Concurrent Aggregates (CA): An Object-Oriented Language for Fine-Grained Message-Passing Machines". Technical Report 1248, MIT Artificial Intelligence Laboratory, July 1990.

[15] D.E. Culler, A. Sah, K.E. Schauser, T. von Eicken, and J. Wawrzynek. "Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine". In *Proceedings of the 4th Conference on Architectural Support for Programming Languages and Systems*, pages 164–175, April 1991.

[16] W.J. Dally, J.A.S. Fiske, J.S. Keen, R.A. Lethin, M.D. Noakes, P. R. Nuth, R.E. Davison, and G.A. Fyler. "The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms". *IEEE Micro*, 12(2):23–39, April 1992.

[17] W.J. Dally, S.W. Keckler, N. Carter, A. Chang, M. Fillo, and W.S. Lee. "M-Machine Architecture v1.0". Concurrent VLSI Architecture Memo 58, MIT Artificial Intelligence Laboratory, January 1994.

[18] R.P. Draves, B.N. Bershad, R.F. Rashid, and R.W. Dean. "Using Continuations to Implement Thread Management and Communication in Operating Systems". In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 122–136, October 1991.

[19] E.W. Felten. "Protocol Compilation: High-Performance Communication for Parallel Programs". Technical Report 93-09-09, University of Washington, September 1993. Ph.D. dissertation.

[20] High Performance Fortran Forum. *"High Performance Fortran Language Specification"*, version 1.0 edition, May 1993.

[21] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. "An Overview of the Fortran D Programming System". In *Proceedings of the 4th Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[22] W. Horwat, A.A. Chien, and W.J. Dally. "Experience with CST: Programming and Implementation". In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 101–109, Portland, OR, June 21–23 1989.

[23] W.C. Hsieh, P. Wang, and W.E. Weihl. "Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems". In *Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming*, pages 239–248, San Diego, CA, May 1993.

[24] V. Karamcheti and A. Chien. "Concert — Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware". In *Proceedings of Supercomputing '93*, Portland, OR, November 15–19, 1993.

[25] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B. Lim. "Integrating Message-Passing and Shared-Memory: Early Experience". In *Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming*, pages 54–63, San Diego, CA, May 1993.

[26] C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynman, M.N. Ganmukhi, J.V. Hill, W.D. Hillis, B.C. Kuszmaul, M.A. St. Pierre, D.S. Wells, M.C. Wong, S. Yang, and R. Zak. "The Network Architecture of the Connection Machine CM-5". Early version appeared in SPAA '92, November 9, 1992.

[27] B. Liskov. "Distributed Programming in Argus". *Communications of the ACM*, 31(3):300–312, March 1988.

[28] F. Mueller. "A Library Implementation of POSIX Threads under UNIX". In *Proceedings of 1993 Winter USENIX*, pages 29–41, San Diego, CA, January 1993. Available at ftp.cs.fsu.edu:/pub/PART/pthreads.tar.Z.

[29] S.K. Reinhardt, J.R. Larus, and D.A. Wood. "Tempest and Typhoon: User-Level Shared Memory". In *Proceedings of the 21st International Symposium on Computer Architecture*, Chicago, IL, April 1994. To appear.

[30] M.C. Rinard, D.J. Scales, and M.S. Lam. "Jade: A High-Level, Machine-Independent Language for Parallel Programming". *IEEE Computer*, 26(6):28–38, June 1993.

[31] D.D. Sleator and R.E. Tarjan. "Amortized Efficiency of List Update and Paging Rules". *Communications of the ACM*, 28(2):202–208, February 1985.

[32] A.Z. Spector. "Performing Remote Operations Efficiently on a Local Computer Network". *Communications of the ACM*, 25(4):246–260, April 1982.

[33] D. Stodolsky, J.B. Chen, and B.N. Bershad. "Fast Interrupt Priority Management in Operating System Kernels". In *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 105–110, San Diego, CA, September 20–21, 1993.

[34] K. Taura. "Design and Implementation of Concurrent Object-Oriented Programming Languages on Stock Multicomputers". Master's thesis, University of Tokyo, February 1994.

[35] C.A. Thekkath and H.M. Levy. "Limits to Low-Latency Communication on High-Speed Networks". *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[36] Thinking Machines Corporation, Cambridge, MA. *"CMMD Reference Manual"*, version 3.0 edition, May 1993.

[37] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. "Active Messages: a Mechanism for Integrated Communication and Computation". In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[38] W. Weihl, E. Brewer, A. Colbrook, C. Dellarocas, W. Hsieh, A. Joseph, C. Waldspurger, and P. Wang. "PRELUDE: A System for Portable Parallel Software". Technical Report MIT/LCS/TR-519, MIT Laboratory for Computer Science, October 1991. Shorter version appears in *Proceedings of PARLE '92*.

[39] J. Wilkes. "Hamlyn — an interface for sender-based communications". Technical Report HPL-OSR-92-13, Hewlett-Packard Laboratories, November 1992.

[40] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. "Vienna Fortran — a Language Specification". Technical Report ICASE Interim Report 21, ICASE NASA Langley Research Center, March 1992.