Optimistic Active Messages: Structuring Systems for High-Performance Communication*

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

MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
USA

{kaashoek,weihl,kerr,wchsieh,tuna}@lcs.mit.edu

Abstract

Recent networks and network interfaces promise remarkable communication performance with very little overhead, but current software structures impose substantial overhead that prevents applications from achieving the benefits of these new architectures. We propose a new software structure that eliminates much of the overhead while preserving the ease of programming of current systems. Our architecture relies on the compiler to bridge the gap between high-level application programs and low-level communication primitives. The compiler incorporates application code into message handlers using a new runtime mechanism called *optimistic active messages*.

*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 an NSF Presidential Young Investigator Award, by IBM, AT&T, and Digital Equipment Corp., 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. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

1 Introduction

Recent networks for parallel and distributed systems promise impressive communication performance, with both high bandwidth and low latency. While the potential for good end-to-end performance is remarkable, reality usually does not live up to that potential: complex layers of system software are interposed between applications and the network, resulting in overhead for protocol processing, data copying, and context switches. Achieving the potential promised by the network hardware requires rethinking the structure of operating systems, runtime systems, and compilers.

The network interface is part of the problem on most current systems: accessing the network from an application program involves going through the operating system, which is usually expensive. Even the best implementations that go through the operating system take hundreds of microseconds (e.g., Thekkath's system takes 170 microseconds for a round-trip remote read operation [14]). Recent designs provide direct user-level access to the network [9, 18, 7, 1, 5, 11]. By integrating protection in the network interface, these proposals offer an order of magnitude performance improvement over previous systems.

Providing user-level access to the network shifts functionality from the operating system to the application. This shift provides the opportunity for the application to optimize its use of the network and the network interface, but also complicates the application writer's job. For example, on the CM-5, applications can use active messages [16, 15] to obtain very high performance, but must worry about deadlock in the network, flow control, reordering of messages, synchronization between messages and computation, and other timing-dependent problems that are handled transparently by lower levels of software in more traditionally structured systems. In distributed systems, the application must also deal with failures in the network and crashes of the sender and the receiver.

The key question is how to deliver the performance of user-level network access while still retaining the relative ease of programming of previous systems. One approach is to provide high-level communication libraries that hide the complexity of dealing with reordering, failures, flow control, deadlock, and synchronization. This approach is fundamentally flawed. It improves on previous systems by avoiding the overhead of entering and exiting the operating system kernel, but there is still substantial overhead to cover various general-purpose but uncommon cases.

We propose a new software architecture that combines ease of programming with very high performance. Users write applications at a high level, and the compiler generates code that accesses the low-level network primitives directly, and usually avoids the synchronization, scheduling, and other overheads incurred in other systems. This shifts the complexity of dealing with communication details onto the compiler, but permits much higher performance to be achieved than is possible with approaches based on high-level communication libraries. The compiler must generate code for the common case (e.g., when messages arrive in order, or when message handlers do not need to wait), but the slower cases are handled by general-purpose libraries.

We have compared our approach to two others: an approach based on a high-level communication library, and hand-coded C programs that use a low-level interface to access the network. The first approach is represented by the Panda/Orca system [3, 4], which we ported to the CM-5. For our approach, we modified the Orca compiler so that it can also generate code that uses the network directly, bypassing the Panda runtime system in the common case. We used a small suite of Orca programs to compare our approach to the Panda/Orca system; for a baseline comparison, we rewrote the programs in C using the active message library [15] on the CM-5. The differences in structure among our approach, using a high-level communication library (as in Panda/Orca), and providing the programmer with a low-level communication library (*e.g.*, C plus active messages) are illustrated in Figure 1.

On a range of microbenchmarks, our approach achieves up to a factor of 30 improvement over Panda/Orca, and is within 25% of the hand-coded programs. On several small applications, our approach results in a factor of 2 improvement in overall application performance, and comes within a factor of 2 of the hand-coded programs.[1]

In the remainder of this paper, we briefly sketch our software architecture, discuss the results of our experiments, and touch on related work.

2 Software Architecture

Existing implementations of high-level parallel languages insulate the programmer from the details of the underlying communication system, but in the process introduce significant overhead. In general, executing a message handler involves many steps: marshaling arguments, sending and

[1] Orca is a higher-level language than C, so part of the performance difference between our approach and hand-coded C programs comes from the overhead of the higher-level mechanisms in Orca (e.g., dynamic arrays) and the lack of aggressive optimization in the Orca compiler.

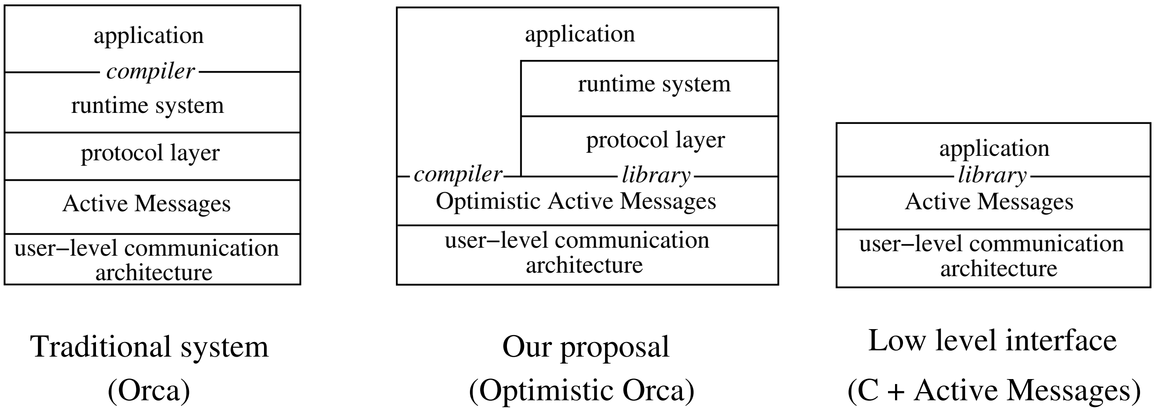

Figure 1: Layers in different message-passing systems. Some of the layer boundaries are library interfaces; others are calls generated by the compiler. The protocol layers implement the RPC and group communication protocols.

receiving the message, creating (or finding) and then context switching to a thread to process the message, unmarshaling the arguments, processing the message, and then a similar sequence of steps to send the reply. All of these steps except for the actual 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 the marshaling and unmarshaling of data.

We propose a new runtime mechanism, called *optimistic active messages* (OAM), that allows application code, including code that needs to acquire a lock or send messages, to execute in a message handler. By pushing more of the application code into message handlers, we can avoid much of the thread management overhead normally associated with processing messages. There are several obstacles that make executing arbitrary code in message handlers 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. Our solution 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 run 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). This approach eliminates thread management overhead for methods that run to completion within a message handler, and will improve performance relative to existing library-based systems as long as most remote method invocations do not block or run for too long.

When a method cannot complete within a message handler, we say that the optimistic execution as an active message must *abort*. In our prototype, there are three reasons why an OAM may have to abort: it needs to wait (e.g., 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. The compiler needs to generate code that detects each of these conditions and, when they occur, transfers control to library routines that handle them.

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 lightweight threads as in Pthreads [10]). Our techniques can be applied to any system that provides similar communication and thread management support.

3 Experiments

To test the feasibility of our approach and to measure its performance benefits, we applied it to Orca. We call this new implementation *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. (Note that our technique can be used with replicated as well as unreplicated objects.) We also compared both Orca implementations with hand-coded C using the active message library [15]; we will use *C+AM* to refer to this implementation. All measurements were taken on a CM-5.[2]

We ran a number of experiments to compare our approach to the others. One microbenchmark, *inc*, measures the time to perform an increment operation on a simple integer object; arguments of various sizes (which are ignored) are sent to measure the effect of data size on performance. The results obtained for *inc* are discussed below; they are typical of the results of our experiments with microbenchmarks, and show that Optimistic Orca performs significantly better than Orca and almost as well as C+AM.

Table 1 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 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 moderate 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.

We also studied three applications used by Bal and Kaashoek to measure the performance of the original Orca system [2]: All-Pairs Shortest Paths (ASP), Successive Overrelaxation (SOR), and the Traveling Salesman Problem (TSP). For these applications, Optimistic Orca is twice as fast as Orca. Optimistic Orca is approximately half as fast as C+AM; since the two systems have similar communication performance, 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 uses an extra data structure through which shared rows are communicated.

We also measured the percentage of message handlers that abort, and found that fewer than 6% message handler executions aborted for each of the applications studied.

4 Related Work

Our work in this paper and other work on applications (e.g., see [6]) indicate that doing significant application-level computation in the message handlers can lead to large improvements in performance. To avoid burdening the programmer with low-level details, we rely on the compiler to bridge the gap.

In the Split-C compiler [16], remote memory

[2]All three systems use Version 3.1 of the CMMD library [15], which implements active messages, running under Patch 1 of CMOST Version 7.2.

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 1: 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.*)

operations are directly compiled into active messages. Split-C, however, is a low-level interface and does not support multithreading. In addition, Split-C supports only a small set of remote operations; additional application-level functionality cannot be embedded into the active message handlers.

The ideas in this paper came in part from our experience with Prelude [17], a high-level parallel programming language. In the Prelude implementation, remote calls incurred significant overhead even though they were typically short and non-blocking. There are many other high-level languages for parallel programming (*e.g.*, Jade [12], Concert [8], ABCL/f [13], and many others); all could make use of optimistic active messages.

The Jade language is implemented on top of a portable runtime system called SAM. The Jade compiler does not compile directly to communications primitives, as we do. Our compilation strategy would allow finer-grained programs to be run efficiently.

In the Concert system, methods that are known to be non-blocking at compile-time are compiled to active messages. 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 is an inflexible and brute-force strategy; we expect that other abort strategies will be more useful at times.

5 Conclusions

New hardware architectures can provide communication with very low overhead, but existing software architectures fail to deliver this performance to applications. *Optimistic active messages* are a new runtime mechanism that eliminates most overhead in the common case, allowing the hardware performance to be delivered to applications while preserving ease of programming. Our approach relies on a change in system structure, shifting functionality into the compiler, which must generate code that detects the uncommon cases and handles them. Experiments show that our approach gives substantial performance improvements over existing approaches. We expect that our approach can also be applied to distributed applications with similar results; we are currently studying the use of our approach in distributed systems and its impact on their structure.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] Frederic T. Chong, Shamik D. Sharma, Eric A. Brewer, and Joel Saltz. Multiprocessor run-time support for fine-grain irregular dags. Technical Report CS-TR-3266, Computer Science Dept., University of Maryland, College Park, March 1994. Submitted for publication.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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*, pages 325–337, Chicago, IL, April 1994.

[12] 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.

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

[14] 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.

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

[16] 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.

[17] 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*.

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