Network Multicomputing Using Recoverable Distributed Shared Memory

John B. Carter*, Alan L. Cox, Sandhya Dwarkadas,
Elmootazbellah N. Elnozahy, David B. Johnson†, Pete Keleher,
Steven Rodrigues, Weimin Yu, and Willy Zwaenepoel

Department of Computer Science
Rice University
Houston, Texas 77251-1892

Abstract

A *network multicomputer* is a multiprocessor in which the processors are connected by general-purpose networking technology, in contrast to current distributed memory multiprocessors where a dedicated special-purpose interconnect is used. The advent of high-speed general-purpose networks provides the impetus for a new look at the network multiprocessor model, by removing the bottleneck of current slow networks. However, major software issues remain unsolved. A convenient machine abstraction must be developed that hides from the application programmer low-level details such as message passing or machine failures. We use *distributed shared memory* as a programming abstraction, and *rollback recovery* through *consistent checkpointing* to provide fault tolerance. Measurements of our implementations of distributed shared memory and consistent checkpointing show that these abstractions can be implemented efficiently.

1 Introduction

In most current distributed memory multicomputers [2] the processors are connected by a dedicated, special-purpose interconnection network, such as a hypercube network or a mesh. In contrast, we are exploring the possibility of building a *network multicomputer* using general-purpose networking technology to interconnect the processors [22]. Such a network multicomputer may be realized as a *processor bank* [32], a number of processors dedicated for the purpose of providing computing cycles. Alternatively, it may consist of a dynamically varying set of machines on which idle cycles are used to perform long-running computations [28]. In either form, such a network multicomputer should be significantly cheaper than current distributed memory multiprocessors since it can be built out of general-purpose commodity technology.

The idea of such a network multicomputer is not new, but its potential has remained largely unrealized. The bandwidth available on general-purpose networks was, until recently, orders of magnitude inferior to the bandwidth provided by special-purpose interconnection networks, such as those present in dedicated multiprocessors. Furthermore, commodity workstations lagged far behind uniprocessor supercomputers in terms of processor speed and floating point support. As a result, it was not uncommon to find that, after months of effort, a carefully parallelized application would run (much) more slowly on a network multicomputer than a sequential implementation of the same application on a conventional supercomputer.

We believe that recent technological breakthroughs have removed, or are about to remove, many of the factors inhibiting network multicomputing. In particular, general-purpose networks with bandwidths in the hundreds of megabits per second are becoming available, and bandwidths in the gigabit range are predicted within a few years. Furthermore, current workstation processors are approaching 100 MIPS and feature much improved floating point hardware. As a result, a much larger class of applications can be supported efficiently on a network multicomputer. It is by no means our position that such loosely coupled multicomputers will render obsolete more tightly coupled designs [7, 24]. In particular, the lower latencies and higher bandwidths of these tightly coupled designs

---

*Current address: Department of Computer Science, University of Utah, Salt Lake City, UT 84112.

†Current address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891.

This work is supported in part by the National Science Foundation under Grants CCR-91163343 and CCR-9211004, and by the Texas Advanced Technology Program under Grant 0036404013. John B. Carter and Pete Keleher were supported by NASA Fellowships. Elmootazbellah N. Elnozahy was supported by an IBM Fellowship.

allow efficient execution of applications with more stringent synchronization and communication requirements.[1] However, we argue that the advances in networking technology and processor performance will greatly expand the class of applications that can be executed efficiently on a network multicomputer.

Although the enabling hardware breakthroughs appear to be on the horizon, many software problems remain to be solved before network multicomputing can become a viable technology. Foremost among these problems is the need for a convenient machine abstraction that eases the burden of parallel programming on a network multicomputer, but at the same time allows efficient execution of a large class of applications. Equally important, this machine abstraction should allow a simple migration path for programs already developed for conventional shared memory multiprocessors. In light of these considerations, we have chosen *distributed shared memory* (DSM) as our programming abstraction. We have built a DSM system, called Munin [10], that provides good performance while requiring only minimal departures from the traditional shared memory model. Sections 2 to 5 describe our approaches to DSM and some of our results.

A machine abstraction for a network multicomputer should also hide one of the most annoying aspects of a distributed system, namely failures. On a general-purpose network with a large number of machines, hardware failures, software crashes, and network partitions present perplexing problems. It is prohibitively expensive in terms of program development cost to expect application programmers to address these problems anew for each application. For this reason we believe that *rollback recovery* using *consistent checkpointing* should be provided as an integral part of the network multicomputer's software. We have chosen this style of fault tolerance because it is transparent — it does not require any effort of the application programmer — and because it provides good performance for typical long-running, noninteractive multicomputer applications. Sections 6 and 7 present our approach and some performance measurements from our implementation of consistent checkpointing.

2 Distributed shared memory

Distributed shared memory (DSM) allows processes to share memory even though they execute on nodes that do not physically share memory [25]. For example, Figure 1 illustrates a DSM system consisting of N separate processors, each with their own memory, connected by a network. DSM provides a more transparent and fine-grained degree

[1] In fact, one of our interests is to also incorporate more tightly-coupled multiprocessors into a network multicomputer.

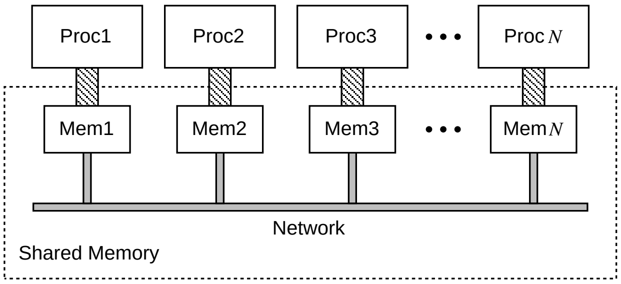

Figure 1 Distributed shared memory

of communication than message passing or remote procedure calls (RPC) [6]. The message passing and RPC approaches relieve the programmer from having to deal with low-level networking details, but data movement must still be programmed explicitly. In contrast, DSM systems move data automatically in response to data access requests by the application. While message passing and RPC are adequate for client-server applications, they lack the desired transparency for parallel programming.

Not only does DSM provide a more convenient programming model, the trend towards fast processors, large memories, and fast networks also holds out the promise of improved DSM performance. Underlying DSM is a *data shipping* paradigm: data is moved to the location performing operations on it. Message passing or RPC often use a *function shipping* paradigm, whereby the operation is moved to the data location. On a low-bandwidth network, the function shipping approach often results in better performance since it is very expensive to move large amounts of data. On higher-bandwidth networks, the cost of data shipping is often negligible when compared to the latency and software overheads involved per message communication. The expense of data shipping can also easily be amortized over multiple accesses by exploiting memory access locality. Modern machines with large main memory sizes provide the ability to cache large portions of the global shared address space, thus allowing a DSM system to aggressively take advantage of the locality of memory accesses. Finally, we anticipate that future computing nodes will be (hardware) shared memory multiprocessors with a small number of processors. DSM appears to be the ideal vehicle for integrating locally shared memory and globally distributed memory.

3 Memory consistency

The provision of memory *consistency* is at the heart of a DSM system: the DSM software must move data among the processors in a manner that provides the illusion of globally shared memory. For instance, in a page-based system, when a page is not present in the local memory of a processor, a page fault occurs. The DSM software brings an up-to-date copy of that page from its remote location into local memory and restarts the process. For example, Figure 2 shows the result of a page fault at processor 1, which results in a copy of the necessary page being retrieved from the local memory of processor 3.

To provide memory consistency *efficiently* is one of the key challenges in building DSM. Three key problems must be addressed. First, sending messages is expensive, and thus the number of messages must be kept low. Sending a message may involve traps into the operating system kernel, interrupts, context switches, and the execution of possibly several layers of networking software. Second, the high latency involved in accessing non-resident memory locations makes it essential to mask the latency of such memory accesses. Finally, the consistency units are large (the size of a virtual memory page), and therefore *false sharing* is a potentially serious problem. False sharing occurs when two or more unrelated data objects are located in the same page and are written concurrently by separate processors, causing the page to *ping-pong* back and forth between the processors. Our approach to DSM provides efficient solutions to address each of these problems.

Early DSM systems have provided consistency by imitating approaches designed for implementing cache coherence in shared memory multiprocessors [25]. We believe that in order to adequately address the problems specific to DSM it is necessary to take a fresh look at consistency implementations for DSM. In particular, we have experimented with novel implementations of relaxed memory consistency models, and we have designed protocols better suited to the needs of DSM. In the next section we describe a prototype DSM system called Munin, which we have built on an Ethernet network of Sun-3/60 workstations. In Section 5, we describe some of the ideas we are experimenting with in our second-generation DSM system.

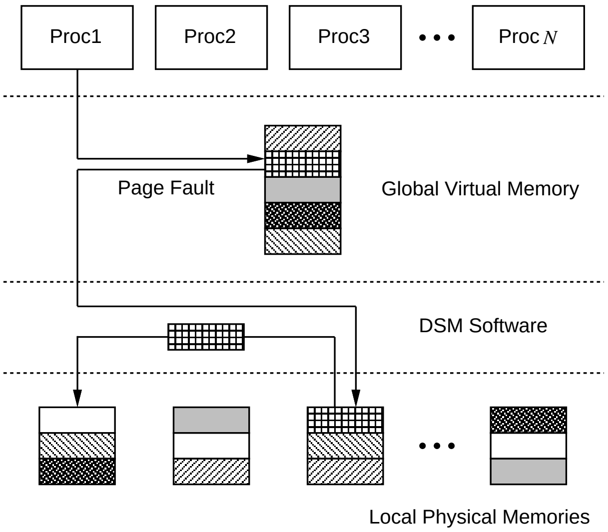

Figure 2 Operation of a page-based DSM system

4 Munin: a prototype DSM system

4.1 Software release consistency

One of the solutions to hiding memory access latency for actively shared data is the use of a relaxed consistency model. Over the past few years, researchers in hardware DSM have adopted relaxed memory consistency models to reduce the *latency* associated with shared memory accesses [1, 13, 16, 27]. In the release consistency (RC) model [16], updates to shared memory must be performed (become visible) only when a subsequent *release* is performed. A *release* in this context can be thought of as a lock release for simplicity, but more sophisticated synchronization mechanisms could also be used. The DASH implementation of RC [16], for example, allows updates to shared memory to be pipelined and overlapped with computation (by allowing reads to bypass writes), thereby reducing latency. Lock acquisition requests for the released lock must, however, be delayed until all previous updates have been performed. For example, Figure 3 shows the pipelined updates sent to processor 2 when processor 1 writes objects x, y, and z under the DASH hardware DSM implementation.

In software DSM systems, it is also important to reduce the *number* of messages exchanged. Therefore, in Munin's software implementation of release consistency [10], updates are not pipelined as in the DASH implementation, but rather are buffered until the release, at which time different updates going to the same destination are merged into a single message. In comparison to Figure 3, Figure 4 shows the same updates to objects x, y, and z merged into a single message sent after the release, using the Munin software DSM implementation.

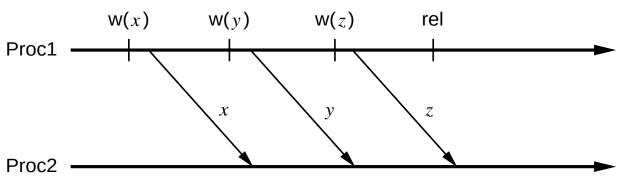

Figure 3 Pipelining of remote memory accesses under DASH RC

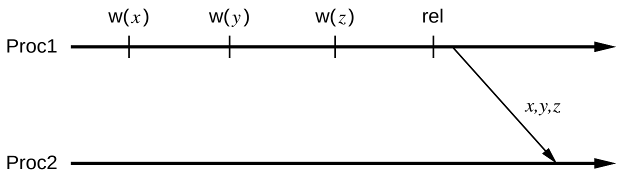

Figure 4 Merging of remote memory updates under Munin RC

4.2 Multiple consistency protocols

In order to further reduce the number of messages exchanged for maintaining consistency, Munin uses multiple consistency protocols, even within a single program execution. Munin allows a separate consistency protocol for each shared object, tuned to the access pattern of that particular object. Munin uses program annotations, provided by the programmer, to choose the consistency protocol parameters for each shared object. Normally, annotations are only needed on object declarations (rather than on each object use), but the programmer may also change the protocol associated with a particular object during the program's execution. These annotations specify the expected access pattern for that particular object, which depends on the program and expected input data.

The system currently recognizes a number of such annotations: *read-only*, *migratory*, *write-shared*, and *conventional*. A *read-only* object avoids the overhead of consistency maintenance and can be replicated. A *migratory* object implies that a single thread performs multiple accesses to the object, including one or more writes, before another thread accesses the object [3, 33]. Such an access pattern is typical of shared objects that are accessed only inside a critical section. The consistency protocol for migratory objects is to migrate the single copy of the object to the new thread, provide it with read and write access (even if the first access is a read), and invalidate the original copy. Compared to a conventional write-invalidate protocol [25], this protocol avoids a write miss and a message to invalidate the old copy when the new thread first modifies the object. A *write-shared* object is written by many processors and is usually the result of false sharing. The consistency protocol exploits the use of release consistency by delaying the updates until a synchronization point (at which point the modifications from different processors are merged), thereby avoiding the *ping-pong* of those pages between processors. A *conventional* object simply uses a conventional write-invalidate protocol [25]. If no annotation is used for some object, that object defaults to *conventional.*

4.3 Performance

Munin was implemented on top of the V kernel [12] on an Ethernet network of Sun-3/60 workstations. A set of library routines linked with the application program, together with some kernel support, forms the core of the Munin system [10]. The system was evaluated by comparing the execution time on Munin of a number of shared memory programs to the execution time of the same applications implemented directly in terms of the underlying message passing primitives of the V kernel. The performance numbers in Table 1 are taken from Carter's Ph.D. dissertation [9], which contains a detailed analysis of the performance of Munin. The table shows the speedup achieved by each application running on 16 processors in each of three cases: using Munin, using a conventional DSM implementation with a single write-invalidate protocol [15, 25], and using message passing.

5 Beyond Munin: lazy release consistency

Munin's implementation of RC may still send more messages than needed for the correct execution of the application. Consider the example of Figure 5, in which processes repeatedly acquire a lock, write the shared object x, and then release the lock. If RC is used in conjunction with an update protocol, and x is present in the caches of all four processors, then these cached copies of x are updated at every release, causing the process that releases the lock to send a message to all other processes. Also, these updates delay the time at which the lock can be released until acknowledgements for all updates have been received. Logically, however, it suffices to update each process's copy only when that process acquires the lock. This problem is not peculiar to the use of an update protocol. Similar examples can be constructed for invalidate protocols. For instance, assume that false sharing exists between objects x and y. The invalidations that are sent at each release after an access to x will cause the entire page, including y, to become invalid. If y is then accessed by another processor, an unnecessary cache miss and reload will occur for that page.

Table 1 Comparison of Munin, conventional DSM, and message passing speedups

Program Name	Munin	Conventional DSM	Message Passing
`matmult`	14.6	14.5	15.6
`grid`	12.3	8.4	12.8
`quicksort`	8.9	3.9	13.4
`tsp`	12.6	11.3	13.2

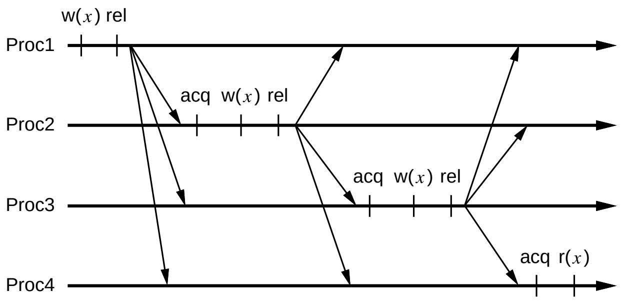

Figure 5 Repeated updates of cached copies under RC

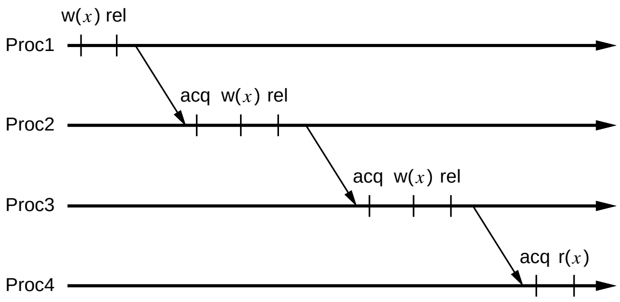

Figure 6 Message traffic under LRC

5.1 Lazy release consistency

Munin attempts to alleviate these problems by using different protocols. In the update protocol example above, the data item x should be annotated as *migratory*. *Lazy release consistency* (LRC) is a new algorithm for implementing the RC model, aimed at reducing both the number of messages and the amount of data exchanged, without requiring such annotations. Unlike *eager* algorithms such as Munin's implementation, LRC does not make modifications globally visible at the time of a release. Instead, LRC guarantees only that a processor that acquires a lock will see all modifications that "precede" the lock acquire. The term "preceding" in this context is to be interpreted in the transitive sense: informally, a modification precedes an acquire if it occurs before any release such that there is a chain of release-acquire operations on the same lock, ending with the current acquire. For instance, in Figure 5, all modifications that occur in program order before any of the releases in processors 1 through 3 precede the lock acquisition in processor 4. With LRC, modifications are propagated at the time of an acquire, and only the modifications that "precede" the acquire are sent to the acquiring processor. The modifications can be piggybacked on the message that grants the lock, further reducing message traffic. Figure 6 shows the message traffic under LRC for the same shared data accesses as in Figure 5. The lock and x are sent in a single message at each acquire.

5.2 Current status

We are currently implementing LRC on SunOS and on Mach-3.0. The SunOS implementation runs on SPARC workstations connected by an Ethernet. Our goal is to develop an efficient implementation running on an unmodified SunOS kernel and using the standard Unix sockets and memory management interfaces. The Mach implementation will run on DECstation workstations connected by an Ethernet and by a 100-megabit per second Fore ATM network. Again, the goal is to use the standard Mach external pager and IPC facilities so that the implementation can be used without kernel changes.

Keleher et al. [20] presents some simulation results showing a notable reduction of message traffic as a result of using LRC compared to RC for the SPLASH benchmark suite [31]. Using a more sophisticated simulator that takes into account software communication latency and network bandwidth, we are currently studying possible speedups. Preliminary results indicate the absolute necessity of using networks with much larger bandwidth than Ethernet, such as ATM, in order to get reasonable speedups with state-of-the-art workstations.

6 Fault tolerance

The need for the programmer to worry about fault tolerance has been another principal inhibitor for workstation multicomputing. Although machine hardware failures are relatively rare, many outages do occur, for example, due to power failures, software crashes, and software maintenance causing the machine to reboot. Furthermore, if the distributed computation is executed as a collection of guest processes on workstations, the return of the workstation's owner may cause processes to be evicted from that machine [28]. If no special precautions are taken, the computation will either "hang" or terminate abnormally, and will need to be restarted from the beginning.

We argue that for parallel programs running on a network multicomputer, fault tolerance should be provided by *transparent* mechanisms, freeing the programmer completely from having to worry about failures. In fact, our implementation of fault tolerance is transparent to the underlying DSM system, and can be used equally well with message passing application programs. Providing transparent fault tolerance also concentrates the code for implementing fault tolerance in a single system module, avoiding needless and error-prone replication of the fault-tolerance support within every application program.

The alternative approach is for the application programmer to deal explicitly with the possibility of failures during program execution. We argue that the complexity of doing so and the attendant program development cost are simply too high. Even for a sequential program this approach is quite burdensome. It requires code for periodically writing the values of key state variables to stable storage during program execution, and code for reading the values of these variables after a failure to allow the program's execution to be continued. Furthermore, the machine reboot procedure must be modified to restart the program after a failure, with the program's recovery routine as the entry point. The problem becomes immensely more complex for a distributed application because a *consistent snapshot* [11] of the application must be saved.[2] For instance, it would be inappropriate to restart a process from a state in which a particular message was received and to restart the sender from a state in which that message had not yet been sent.

A more structured approach to fault tolerance is provided by mechanisms such as *recovery blocks* [18], *transactions* [17], or reliable broadcasting facilities [5]. These systems provide the application programmer with a set of basic primitives on which to build fault tolerance. While these approaches certainly have merit for other application areas, for parallel programming they present too much of a burden on the application programmer. Furthermore, as we will show in Section 7, transparent methods cause only very minor performance degradation, calling into question the need for application-specific techniques.

7 Consistent checkpointing

Consistent checkpointing is an attractive approach for transparently adding fault tolerance to distributed applications [14, 21]. With consistent checkpointing, the state of each process is saved separately on stable storage as a *process checkpoint*, and the checkpointing of individual processes is synchronized such that the collection of checkpoints represents a *consistent state* of the whole system [11]. A set of checkpoints records a consistent state if all messages recorded in the checkpoint as having been received are also recorded as having been sent in the state of the sender. For example, the system state indicated by the first dotted line in Figure 7 is inconsistent, whereas the second system state shown is consistent. After a failure, failed processes are restarted on any available machine and their address space is restored from their latest checkpoint on stable storage. Surviving processes may also have to roll back to their latest checkpoint on stable storage in order to remain consistent with recovering processes [21].

[2]The term *consistency* is used here with a different meaning than when discussing distributed shared memory earlier in the paper. We will continue to use the word *consistency* because it is standard terminology in the fault-tolerance literature.

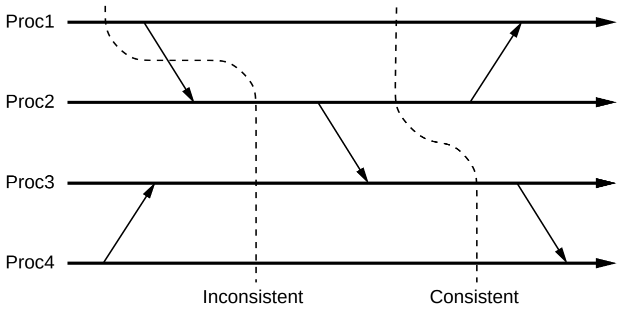

Figure 7 Consistent and inconsistent system states

7.1 Implementation

Many consistent checkpointing protocols have appeared in the literature (e.g., [11, 21]). In our protocol [14], each consistent checkpoint is identified by a monotonically increasing Consistent Checkpoint Number *(CCN)*. One distinguished process acts as a *coordinator*. In the first phase of the protocol, the *coordinator* starts a new consistent checkpoint by incrementing the *CCN* and sending *marker* messages [11] that contain the *CCN* to all other processes. Upon receiving a *marker* message, a process takes a tentative checkpoint. Furthermore, every application message is tagged with the *CCN* of its sender [8, 23]. A process also takes a tentative checkpoint if it receives an *application* message whose appended *CCN* is greater than the local *CCN*. The resulting checkpoints form a consistent state. Figure 8 shows an example execution of the first phase of the protocol, in which a system of three processes take consistent checkpoint number 9. In the second phase of the protocol, processes inform the coordinator that they have taken the checkpoint, and the coordinator then instructs all processes to make their tentative checkpoint permanent and to delete the previous checkpoint.

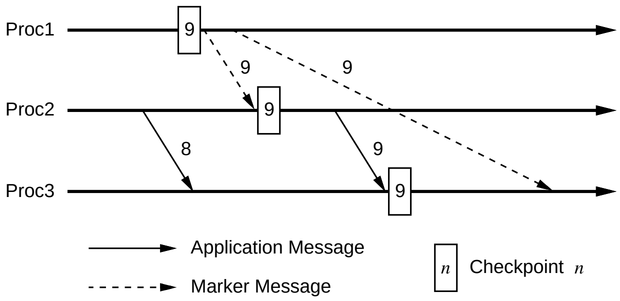

Figure 8 Consistent checkpointing protocol

The coordination of the different checkpoints so that they form a consistent state entails relatively little overhead. The key to efficiency in checkpointing is to avoid interference between the execution of the process and the recording of its checkpoint. We use two techniques, *incremental* checkpointing [14, 19] and nonblocking *copy-on-write* checkpointing [26], to reduce this interference. Incremental checkpointing writes to stable storage only those pages of the address space that have been modified since the previous checkpoint. The set of pages to be written is determined using the *dirty* bit maintained by the memory management hardware in each page table entry. Copy-on-write checkpointing allows the application to continue executing while its checkpoint is being written to stable storage. Copy-on-write memory protection is used to prevent a process from overwriting part of its address space before it is written to the checkpoint.

The fault-tolerance support is transparent to the DSM support, and we have thus been able to implement the two independently. Both implementations, though, involved modifications to the V kernel [12], and we have not yet integrated the two into a single kernel. We have therefore not yet been able to measure the overhead of consistent checkpointing for shared memory applications, but we believe that it will be similar to that for message passing applications. In effect, the DSM support and the shared memory application program together act as a message passing application on top of the consistent checkpointing implementation.

In addition, we are exploring a number of areas in which the DSM support can provide assistance for improving the performance of consistent checkpointing. For example, when copies of a *read-only* object are replicated, the hardware *dirty* bit in the page table entry for the corresponding page at each destination processor could be turned off at the same time as the page is made read-only, preventing the copy-on-write checkpointing from writing that page to stable storage on the next checkpoint of that processor. A similar opportunity arises, for example, when a *migratory* object is moved from one processor to another. If there are no other objects in the page from which the *migratory* object is being moved, the *dirty* bit in that page table entry could be turned off to prevent that page from being written to stable storage on the next checkpoint of that processor; the object will instead be written to stable storage as a part of the checkpoint of the processor to which the object was moved. No object can mistakenly be left out of a checkpoint as a result of such optimizations performed by the DSM support, since each checkpoint records a consistent state of the application program, with respect to the messages sent and received by the application program *and* those sent and received by the DSM support.

7.2 Performance

We have measured the performance of our implementation of consistent checkpointing on an Ethernet network of 16 Sun-3/60 workstations. The results demonstrate that consistent checkpointing is an efficient approach for providing fault-tolerance for long-running distributed applications. Table 2, taken from Elnozahy et al. [14], shows the increase in the running times of eight large, message passing application programs relative to the running times for the same programs without checkpointing. In these experiments, checkpoints were taken every 2 minutes. Even with this small checkpoint interval, consistent checkpointing on average increased the running time of the applications by only 1%. The worst overhead measured was 5.8%.

Elnozahy et al. [14] presents a detailed analysis of the measurements of the performance of consistent checkpointing, which further demonstrates the benefits of nonblocking copy-on-write checkpointing and incremental checkpointing. Copy-on-write checkpointing avoids a high penalty for checkpointing for processes with large checkpoints, a penalty that reached as high as 85% for one of our applications. Using incremental checkpointing reduces the load on the stable storage server and the impact of the checkpointing on the execution of the program. Without incremental checkpointing, the worst overhead measured for any application increased from 5.8% to 17%. Synchronizing the checkpoints to form a consistent checkpoint increased the running time of the applications by very little, 3% at most, compared to independent checkpointing with no synchronization [4]. In return, consistent checkpointing limits rollback to the most recent consistent checkpoint, avoids the domino effect [29, 30], and does not require garbage collection of obsolete checkpoints.

Table 2 Comparison of running times with and without checkpointing

Program Name	Without Checkp. (sec.)	With Checkp. (sec.)	Difference (sec.)	Difference %
`fft`	11157	11184	27	0.2
`gauss`	2875	2885	10	0.3
`grid`	3552	3618	66	1.8
`matmult`	8203	8219	16	0.2
`nqueens`	4600	4600	0	0.0
`primes`	3181	3193	12	0.4
`sparse`	3893	4119	226	5.8
`tsp`	4362	4362	0	0.0

8 Concluding remarks

Technological breakthroughs, especially in the area of high-speed networking, allow us to design a multicomputer using general-purpose networking technology with the attendant benefits in cost compared to dedicated interconnection networks. Major software problems remain to be resolved before such a network multicomputer can become practical. In particular, application programmers should not be expected to deal with low-level message passing or with recovery from failures. We use *distributed shared memory* to hide the message passing, and *consistent checkpointing* to recover from failures. We have shown that by using novel implementations of consistency models and protocols, DSM systems with good performance can be built. Furthermore, we have shown that through the use of nonblocking, incremental checkpointing, consistent checkpointing adds very little overhead to the program's execution.

References

[1] S. Adve and M. Hill. Weak ordering: A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] W.C. Athas and C.L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, 21(8), August 1988.

[3] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 125–134, May 1990.

[4] B. Bhargava and S-R. Lian. Independent checkpointing and concurrent rollback recovery for distributed systems — an optimistic approach. In *Proceedings of the 7th Symposium on Reliable Distributed Systems*, pages 3–12, October 1988.

[5] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[6] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[7] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H.T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P.S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An integrated solution to high-speed parallel computing. In *Proceedings Supercomputing '88*, pages 330–339, November 1988.

[8] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Proceedings of the 4th Symposium on Reliable Distributed Systems*, pages 207–215, October 1984.

[9] J.B. Carter. *Munin: Efficient Distributed Shared Memory Using Multi-Protocol Release Consistency*. PhD thesis, Rice University, December 1992.

[10] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[11] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[12] D.R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[13] M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Computers*, 16(6):660–673, June 1990.

[14] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[15] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, December 1989.

[16] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, May 1990.

[17] J.N. Gray. Notes on database operating systems. In R. Bayer, R.M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[18] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In E. Gelenbe and C. Kaiser, editors, *Operating Systems*, volume 16 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, 1974.

[19] D.B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. PhD thesis, Rice University, December 1989.

[20] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[21] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[22] H.T. Kung, R. Sansom, S. Schlick, P. Steenkiste, M. Arnould, F.J. Bitz, F. Christianson, E.C. Cooper, O. Menzilcioglu, D. Ombres, and B. Zill. Network-based multicomputers: An emerging parallel architecture. In *Proceedings Supercomputing '91*, November 1991.

[23] T.H. Lai and T.H. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, May 1987.

[24] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[25] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[26] K. Li, J.F. Naughton, and J.S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming*, pages 79–88, March 1990.

[27] R.J. Lipton and J.S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[28] M. Litzkow, M. Livny, and M. Mutka. Condor — a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, June 1988.

[29] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[30] D.L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, March 1980.

[31] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[32] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[33] W.-D. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.