Predicting the performance of distributed virtual shared-memory applications

by E. W. Parsons
M. Brorsson
K. C. Sevcik

The use of networks of workstations for parallel computing is becoming increasingly common. Networks of workstations are attractive for a large class of parallel applications that can tolerate the higher network latencies and lower bandwidth associated with commodity networks. Several software packages, such as TreadMarks™, have been developed to provide a common view of global memory, allowing many shared-memory parallel applications to be easily ported to networks of workstations. This paper investigates in detail the performance of several TreadMarks-based shared-memory applications on a modern network of workstations, identifying the extent to which different system components affect the efficiency of these applications. Then a performance model for such applications is developed and used to evaluate the impact future changes in technology are likely to have on performance. The results of the model indicate that current systems are limited in their performance by communications and software overhead for supporting the distributed virtual shared memory, rather than hardware delays.

Even though most parallel computing today is based on a message-passing programming model, it is easier to develop parallel programs using a *shared-memory* image for interprocess communication. In fact, some algorithms are extremely difficult to parallelize by hand using explicit message passing.[1] As a result, much research has been devoted to presenting a shared-memory image to programs running on distributed-memory systems, an approach termed *distributed virtual shared memory* (DVSM). This is accomplished using a combination of virtual memory protection mechanisms to detect accesses to specific portions of memory, and software exception handlers to ensure that the memory images on different processors are kept consistent. Examples of such software-based DVSM systems are TreadMarks**, CVM, Munin, and Ivy.

It is becoming increasingly common for networks of workstations (NOWs), instead of special-purpose multiprocessors, to be used for parallel computation, primarily because of the cost-effectiveness of using commodity components. Also, the latest processor technology often appears in workstations before it can be incorporated into large-scale multiprocessors, which allows NOWs to attain higher aggregate performance relative to a large-scale multiprocessor for the same investment. This fact can be seen in that several recent winners of the Gordon Bell Prize in the best cost/performance category have used a NOW as their computing platform.[2–4]

In most such systems, the granularity for memory protection is relatively coarse, namely the page size. With straightforward implementations of DVSM, this granularity leads to *false sharing*. This effect occurs when one processor updates a data item in a memory page, causing update or invalidation messages to be sent to all other processors that have copies of that page, even though those processors never access the particular data item just updated. With use of relaxed forms of memory consistency, it is possible to allow several processes to write to a particular page simultaneously, without causing excessive exchanges of messages. With such forms of consistency, reasonably good speedups can be obtained for parallel applications, even on a NOW using off-the-shelf LAN (local area network) technology.[5]

However, given that processor performance is increasing very rapidly relative to commodity network performance, it is not clear how well such DVSM systems will perform in the future. Whereas significant speedups (e.g., about three for the Barnes applications, defined later) were reported for certain applications on eight processors under the TreadMarks system,[6] those same applications exhibited about half the speedup running on more recent processors connected by the same network.

In this paper, we describe the way in which we developed a model to predict the performance of parallel DVSM-based applications, using TreadMarks as a case study. The purpose of this model is to allow us to determine the effects of changes in technology on the performance of applications. As an abstraction of a system, a model does not include complete details about every aspect of the system (e.g., the memory system in our case). Despite this, a model can often offer good approximations to performance given certain types of changes. For example, we use this model to investigate the impact of processor speed, network latency and bandwidth, and software overheads on performance. Although this study is based on applications using TreadMarks, we believe the approach we describe is generally applicable to the modelling of applications based on other DVSM systems as well.

The results show, not surprisingly, that consistency actions and lock acquisitions are the limiting factors on performance with current technology. When we break down the costs associated with these operations into software delays (DVSM and communication protocols processing time) and hardware delays (network time and adapter latency), we find that the hardware delays do not constrain the performance as much as the software delays. However, hardware delays will become increasingly significant as processing speeds increase. The results also indicate that aggressive network hardware technologies by themselves are not enough to achieve the same performance for systems with processors eight times faster than today's; in particular, latency hiding techniques (e.g., prefetching) will be required to maintain levels of processor efficiency comparable to those achieved now.

The next section discusses a specific DVSM system, TreadMarks, and the technology parameters that affect the performance of applications that run under it. The succeeding section presents performance results for a number of applications running on our experimental system using TreadMarks. Then the model that evaluates the performance of these applications is presented, along with a validation of the execution times predicted by our model relative to those actually observed. The model is then used to predict future performance limitations. Related work and conclusions are described in the last two sections, respectively.

Technology parameters affecting performance of DVSM

There have been several experimental implementations of DVSM systems, all of which face the same fundamental challenge of maintaining consistency of shared data without encountering an unacceptable amount of synchronisation and communication overhead. The specific DVSM system we have chosen to study is the TreadMarks system. Developed for a network of workstations, this system allows parallel programs to interact transparently, using a shared-memory model, even though the processors themselves do not physically share memory. TreadMarks, like its similar predecessors, does this by relying on the memory protection mechanism provided by the hardware and operating system to detect accesses to specific regions of memory in each processor, and then invoking necessary actions to maintain memory consistency. These actions involve exchanging messages between processors to update the contents of memory.

Lazy release consistency. In a strong memory consistency model, a programmer can assume that modifications made to memory in a shared-memory segment are automatically and immediately reflected in the memories of other processors. This model

would correspond to the behaviour of multiple processes sharing a memory segment on a single workstation (possibly having multiple processors). This model, however, tends to require a large number of message exchanges when implemented on distributed-memory machines.

As a result, a number of weak consistency memory models have been proposed that greatly reduce the amount of interprocess communication. They are based on the observation that most parallel programs serialize modifications to a given data item from multiple processors through the use of locks. In such programs, it is possible to reflect modifications made to memory only when a lock is released while preserving the correctness of the algorithm (since no other process should be reading from or writing to this memory item while the lock is held). In fact, in this model, a processor only needs to be informed of modifications to a data item when it tries to acquire the lock protecting the data. This condition allows different processors to simultaneously modify different variables that lie on the same page, as long as no two processors modify the same variable at the same time.

The lazy release consistency (LRC) protocol[7] used in TreadMarks is very similar to the basic release consistency model just described, except that data are simply marked as having been modified upon lock acquisition, delaying the application of the modifications until the data are actually accessed. As this protocol has been described numerous times in the past, we do not repeat this description here. Instead, we provide an example of the protocol in the Appendix, where a number of terms are described in detail. Briefly, a *fault* occurs when a processor accesses a page that requires some type of consistency action, resulting from the processor having received prior *write notices* for the page. To make the page consistent, the processor requests *diff* from other processors that have modified the page.

Performance factors. The original TreadMarks study reported reasonable speedups over a range of applications on a network of eight DECstation**-5000/240 workstations interconnected by a Fore ATM (asynchronous transfer mode) switch.[6] Since that study was done, processor performance has increased dramatically while the ATM network technology still is considered to be current. In the future, the performance of DVSM-based applications will be strongly influenced by relative changes in the performance of these two components, as well as by developments in system software and compiler technology. Within this subsection some of the factors that might influence the performance of DVSM applications in the future are discussed.

Processor performance. Processor performance continues to increase roughly by a factor of two every 18 months. If this rate of increasing performance is not matched in other system components, it will be difficult to maintain current levels of performance for parallel applications, as measured by *speedup* or, equivalently, *efficiency*. (Speedup is the ratio of execution time on a single processor over that on multiple processors, and efficiency is the speedup divided by number of processors.) The reason is that, as the computation time decreases, the overheads due to processor interactions will represent a larger fraction of the overall execution time.

TreadMarks allows parallel programs to interact transparently, using a shared-memory model.

If we consider the system used in the study by Keleher[6] and Amza et al.,[5] the processors represent technology that is now five years old. If other parameters were to exhibit the same rate of improvement, commodity network bandwidth would have to increase from 155 Mbps (megabits per second) to about 1 Gbps (gigabit per second), and network latency would have to drop from about 500 μs (microseconds) to 100 μs for a one-way message. Even though there are no technical reasons preventing network components from keeping pace with advances in processor technology, development efforts for commodity network components tend to focus on increasing bandwidth rather than on reducing latency, the latter being most important to DVSM systems.

Network performance. Two types of off-the-shelf LAN technologies that are of interest in a NOW are Ethernet and ATM. The 10-Mbps Ethernet, which has been very common up to now, is quickly being displaced by 100-Mbps Fast Ethernet, as the latter is

standard in many workstations sold today. This technology offers much greater bandwidth, resulting in lower contention and shorter delivery time for large messages, relative to its predecessor. The ATM networks used today, in contrast, typically operate at 155 Mbps, but these will soon be displaced by 622-Mbps networks now emerging on the market. Because it uses a switch, an ATM network will incur less contention than a Fast Ethernet network for independent communications at the expense of added latency through the switch. This difference disappears, however, if one considers using switched Fast Ethernet hubs, which are becoming increasingly popular.

Network adapters are becoming increasingly capable in terms of the services they can provide. An example is the Cheetah ATM network adapter, developed by IBM.[8] This adapter has the ability to read from and write to user-specified data buffers, thus avoiding having to copy messages to and from the kernel as has traditionally been the case. Experiments with these adapters on our system have shown that latencies for small messages of 140 μs can be achieved (as compared to 250 μs for standard Transmission Control Protocol/Internet Protocol, or TCP/IP, stacks). Others have reported even lower latencies using experimental interfaces or protocols (e.g., References 9 and 10).

System software. TreadMarks is but one example of a DVSM system, albeit the one that is most common. As we gain more experience with such software, the overheads associated with DVSM will decrease as more efficient algorithms and data structures are developed. Also, since communication latency due to protocol processing is becoming more significant, future systems are likely to provide lightweight protocols for the types of applications examined in this paper.

Compiler technology. One of the primary performance limitations for DVSM-based applications is the latency for consistency actions and lock acquisitions. Research in compiler technology has devised techniques to hide the latency of cache misses, but these same techniques have been shown to be highly applicable to paging for out-of-core computations.[11] Using such techniques to prefetch diffs for pages that will be accessed shortly[12] or to acquire locks in advance of when they are needed[13] may greatly improve the performance of applications. Hiding latencies will likely become more important in the future as it is generally easier to increase bandwidth than to reduce latency.

Performance analysis of TreadMarks on a network of workstations

This section describes our experimental system and the performance results for some applications that run on it using TreadMarks.

Experimental platform. Our experimental platform consists of eight IBM RISC System/6000* 43P (133 MHz) workstations, each having 64 MB (megabytes) of main memory, and connected by two commodity networks, a 100 Mbps Fast Ethernet and a 155 Mbps ATM. The Fast Ethernet comprises 3Com 3C595 EtherLinkIII** PCI (peripheral component interconnect) adapters and a 16-port Cisco hub, whereas Fore PCA200E ATM adapters and a Fore ASX 200/WG ATM switch comprise the ATM network.

Choice of applications. We have chosen six applications to analyze for this study, including ones that do numerical computation, image analysis, and physical systems modelling. These applications are:

SOR—This kernel performs a typical red-black successive over-relaxation (SOR) on a two-dimensional grid, which involves iteratively updating each element of the grid based on the values of its neighboring elements. The grid is partitioned across processors in contiguous areas, so communication only occurs when a processor must read an element across a boundary; synchronisation is based on a barrier at the end of each iteration.

IS—This kernel sorts 2^N integers in the range from zero to $2^B - 1$, using a bucket sort algorithm. Each iteration in the algorithm consists of two steps. In the first step, the data-sharing pattern is migratory, whereas in the second, it is primarily read-only.

Barnes—This application simulates the evolution of a system of bodies under the influence of gravitational forces (e.g., a system of galaxies). The iterative algorithm consists of phases with a producer-consumer data-sharing pattern between processors in different phases. This sharing is relatively fine-grained, causing considerable false sharing and, hence, high fault handling overhead.

Sphere—The shallow water equations for a spherical surface are solved by this application. The algorithm used in Sphere is typical of those used in

Table 1 Summary of application characteristics

Application	Parameters	Sequential Exe. Time(s)	Parallel Execution Time (8 processors)			
			Fast Ethernet		ATM	
			Exe. Time	Speedup	Exe. Time	Speedup
SOR	2000 × 1000 doubles	10.3	1.5	6.9	1.5	6.9
IS	2^{23} ints, 2^{9} buckets	6.1	1.1	5.5	1.1	5.5
Barnes (1)	16 384 particles, tolerance 0.57	254.8	70.6	3.6	67.5	3.8
Barnes (2)	32 768 particles, tolerance 0.67	413.4	124.3	3.3	119.9	3.4
Sphere (1)	pitch 32	93.5	25.0	3.7	23.2	4.0
Sphere (2)	pitch 64	351.0	77.7	4.5	78.8	4.5
Water	1000 molecules	119.0	38.7	3.1	38.1	3.1
Raytrace	"balls4" image	154.0	27.4	5.6	27.0	5.7

climate and weather modelling software.[14] Almost all shared-data structures are initialised at the beginning and then accessed in a read-only fashion throughout the remainder of the execution. The one exception is a solution vector whose elements can be updated by any processor while the equations are being solved.

Water—This application simulates a system of water molecules in a liquid state. The main shared-data structure is an array that is accessed by processors in a partitioned manner. The primary sharing pattern is fine-grained and migratory, as intermolecular forces are calculated, but some false sharing occurs across the boundaries of the array.

Raytrace—This application renders a two-dimensional graphical image created from a three-dimensional scene, using a raytrace algorithm. Almost all data are shared read-only or exclusively updated by one processor as each processor operates on its own partition of the frame buffer used to store the resulting image. However, for load-balancing reasons, processors may steal work from one another, causing some degree of false sharing.

Table 1 lists the parameters used for each application and summarizes their execution-time performance on our experimental platform, based on the average of five trials for each case. The data sets that we have used are in many cases representative of production runs of the programs. However, many of the applications use iterative algorithms, and, in order to reduce the length of experiments, we have limited the number of iterations to only a few. We therefore deliberately analysed only the parallel component of each application, since the serial initialisation would otherwise have had too great an impact on our results. For Barnes and Sphere, two different problem sizes, determined by "particles" and "pitch," respectively, are included in our experiments. (In the case of Barnes, the tolerance differs between the two problem sizes, as recommended by Singh et al.[15] in order to achieve the same relative error.)

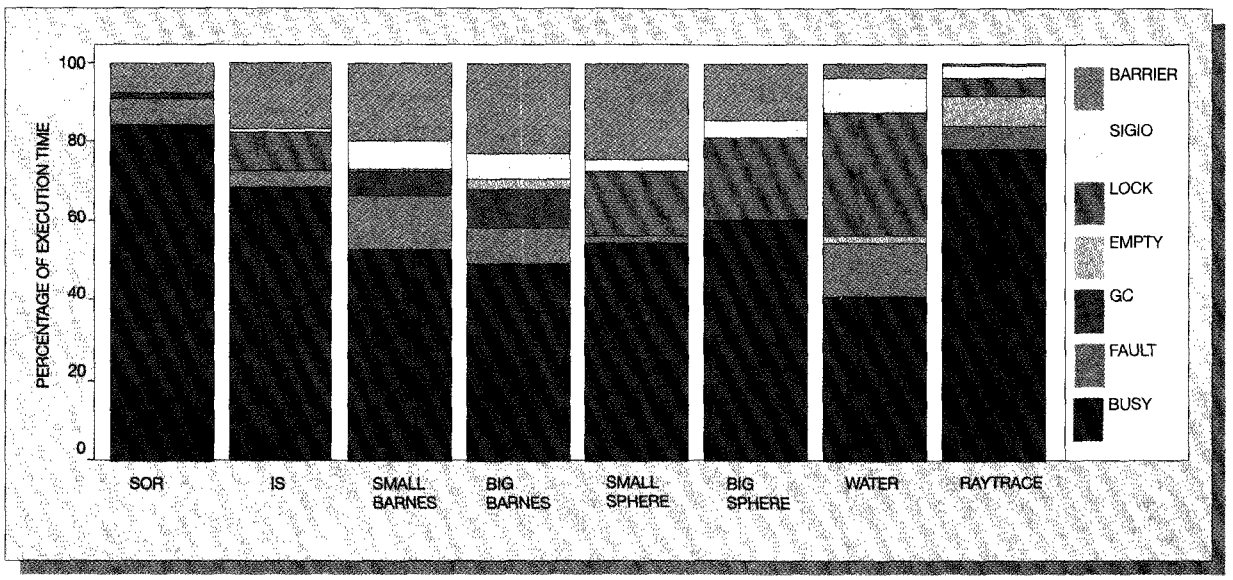
Execution time overhead. We instrumented TreadMarks to measure the amount of time spent in different components, as follows:

Busy time—Busy time is the time spent by the application on actual computation.

Fault handling—This time is consumed by handling read or write faults of the application to maintain memory consistency across processors. The most significant component of fault handling is the sending of requests to remote processors for (possibly multiple) diffs, and waiting for their replies.

Empty time—This time is consumed in handling first-time misses for pages accessed by each processor. Before consistency action can be taken, a full copy

Figure 1 The percentage of execution time spent in executing useful instructions and various overhead components



of the page must be obtained from another processor (known as an empty miss). Although empty misses occur as a result of a consistency fault, they occur with varying frequencies in different applications, so they are modelled separately.

GC time—This time is consumed by garbage collection, which is initiated at the end of a barrier if the amount of memory consumed by TreadMarks data structures exceeds a predefined threshold. Garbage collection results in essentially the same operations as a consistency fault, requesting and receiving diffs, except it does so for all pages being actively shared. As such, it is an expensive operation, because two processors may exchange diffs during the GC phase even if their sharing patterns do not require such exchanges.

Lock acquisition and release time—This time is consumed in acquiring and releasing locks, which are typically used to serialize access to shared data. Acquiring a lock involves sending a request to the lock manager, which forwards the request to the current holder of the lock. Releasing a lock typically does not require any messaging, unless another processor is already waiting for the lock.

Sigio time—This time is consumed by the handling of asynchronous I/O requests (i.e., SIGIO in UNIX**) from remote processors resulting from DVSM activities (i.e., barriers, faults, or locks). This component differs from the previous ones in that it is not initiated by the local processor, but is rather an overhead imposed by remote processors.

Barrier time—This time is consumed by barrier operations, which are used to synchronise all processors involved in a computation at the same point. Apart from the delay waiting for all processors to arrive at the barrier, the principal cost of a barrier is all the memory protection systems calls that must be made to the kernel to invalidate pages as a result of the combined write notices from all processors.

The results for each application are shown in Figure 1. As can be seen, DVSM overheads can be quite significant, accounting for up to 60 percent of the overall execution time in the case of Water. In general, the high cost observed for barriers is not due to messaging, but rather due to imbalance between processors, which causes significant wait times. In many cases, this imbalance arises from different processors having different amounts of DVSM overheads, rather than from imbalance internal to the algorithm. IS, Sphere, and Water all have a large amount of locking overhead, whereas Barnes suffers mostly from garbage collection and fault-handling overhead.

Performance model of applications

In the previous section, we presented time spent in different components for each of our six applications. Next, we describe a model that we used to study the effects of technology changes on the magnitude of each of these components. Our basic approach to developing this model is to break down the cost of each significant operation in each component. In particular, we separate the time spent in local and remote computation and in all major parts of the communication paths. We also take into account the contention that occurs at local and remote processors resulting from DVSM activities.

To obtain our measurements, we used a lightweight timing facility in the Advanced Interactive Executive* (AIX*) to measure elapsed time between various points in the TreadMarks software. Since we ran all our experiments while the system was quiescent, the effect of daemon activity and interrupts unrelated to the applications is negligible (as is supported by the repeatability of our experiments). All applications were small enough to fit into memory, so there was very little paging activity.

One assumption in our model is that network contention at a global level is negligible, because either a switched network is being used (e.g., an ATM switch) or the available network bandwidth is very high. In particular, we found that network contention was quite low even for our nonswitched Fast Ethernet network. We do, however, take into account the contention that arises between a processor and the network, a problem that can increase the cost of certain DVSM operations.

Given the nature of our model, we do not attempt to characterize memory system performance, and instead assume that memory access costs are in most cases unaffected by the changes in technology that we examine in the next section. It is possible, however, to adjust the performance of the memory system at a global level, as we have done for the case of increasing processor speeds in the second subsection of the next section. Assessing the effects of memory at a more detailed level can only be fully achieved using low-level simulation techniques.

DVSM overheads. Ideally, the time spent in each of the execution time components (as described previously) would be evenly balanced across all processors. As mentioned above, we have found that considerable imbalance can exist in several of these, which can potentially lead to a large modelling error if not taken into account. For example, in Barnes, there are a significant number of faults, but the source of these faults alternates between processor zero and all other processors from one phase of the computation to the next. Averaging the faults across the entire computation would lead to large inaccuracies in the model.

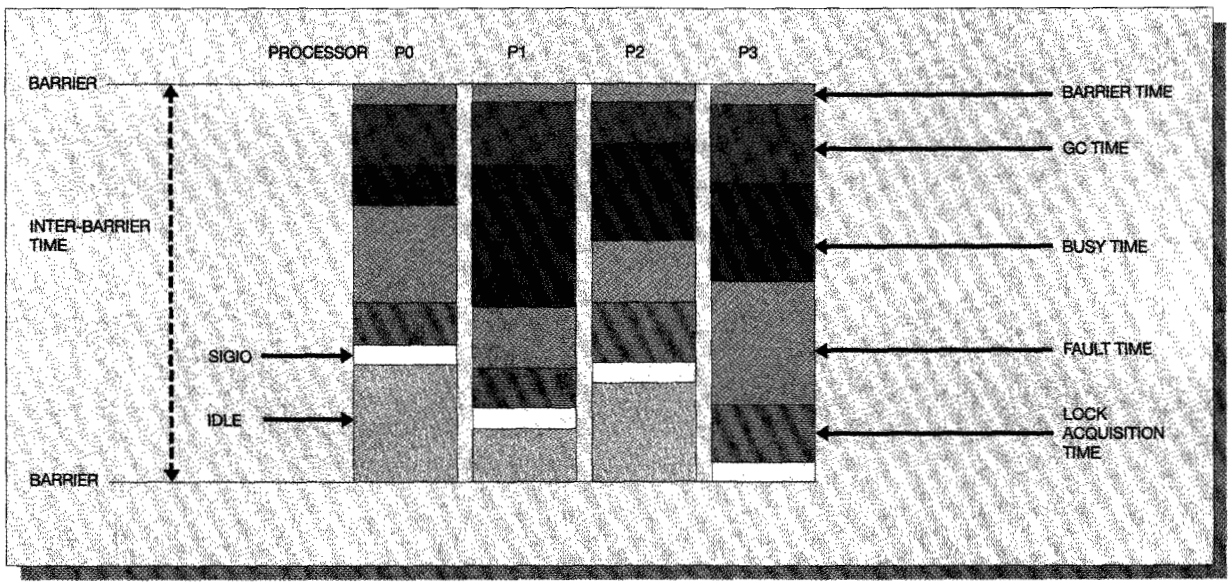
To model the execution time of an application, we consider separately each phase of the computation, as defined by the barriers. For each phase, we first determine the total amount of time spent in each component (busy time or DVSM overhead), whose sum represents the nonidle time for the processor; the time required for the phase is simply the maximum of these sums. (As Figure 2 illustrates, some processors may be idle at the end of a phase, waiting for the slowest to arrive at the barrier. Note that each processor may be in a particular component many times during each phase, which is represented by a single "segment" in the graph.) The total execution time for the application is given by the sum of these maximums over all phases of the computation.

To determine the time required by each component, we instrumented TreadMarks to collect, for each type of operation and each phase of the computation, (1) the number of operations occurring on each processor, (2) the average time spent in software, both on local and remote processors, and (3) the sizes of average messages involved in any interactions. In addition, we observed that requests interrupting remote processors can be delayed for several reasons:

- I/O interrupts may be temporarily disabled if the remote processor is already busy with some other TreadMarks operation, causing the request to be delayed until interrupts are re-enabled.
- The remote processor may have several outstanding requests, behind which the current request must wait.
- The remote processor may be operating in kernel mode, either because it experienced a page fault (possibly requiring a lengthy disk access) or because it made a system call.

Since these delays can be quite significant, we also measure the average time remote operations have to wait as a result of the first two types of delay; we were unable to measure the last because that would have required access to kernel source code.

Figure 2 Components of parallel execution time between barriers. (To estimate the time for a phase, we sum the estimated times required for each component on each processor and take the maximum value.)



We then input these data into a spreadsheet that computes, based on the measurements, the time spent by each processor in each component during each phase, using these to estimate the overall computation time for the application. By breaking down the cost of each operation (as illustrated later), we can then predict the effects changes in technology will have on the performance of the application.

Next, we provide more detail about the way the faults and barriers are treated, as these represent the more complex of the components.

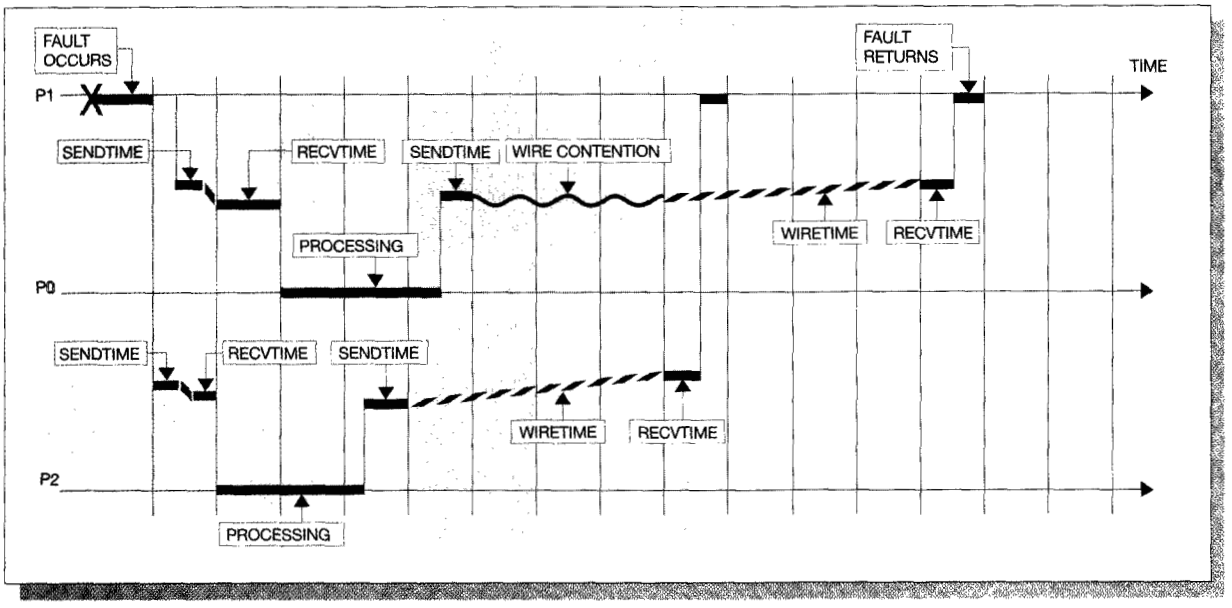
Fault time. The major cost of handling a fault is communicating with remote processors to obtain any necessary diffs for a page. Modelling a fault is the most difficult aspect of our model, particularly when diffs must be obtained from several processors.

To illustrate this complexity, consider the case where a processor (P1) must acquire diffs from two other processors (P0 and P2) as a result of a fault. (A similar but extended example illustrating the need for multipart diff requests is presented in the Appendix.) The details of such an operation are illustrated in Figure 3. When the fault occurs, a diff request message is first sent to the two remote processors, each of which computes and returns a diff in parallel. Since requests are very small (12 bytes), there is no significant wire time in the sending of requests, but it is quite possible for responses to be as large as 4 KB in size, corresponding to wire times of about 330 μs on a Fast Ethernet.

Essentially, a multipart diff request is a pipelined operation where (1) the sending of the requests, (2) the receipt of the replies, and (3) the time spent on the wire must be serialized. In our model, we thus estimate the cost of each of these three components and choose the longest in estimating the overall cost of a multipart diff request.

Barrier time. The barrier time for slave processors (i.e., processors other than the master of the barrier) consists of three phases. First, write notices are created. Second, a synchronous request is then sent to the master, blocking the slave until the master releases the barrier. Finally, memory pages are invalidated according to the write notices that are sent by the master. For the master processor, further processing occurs in receiving barrier request messages from slave processors, merging write notices, and sending release messages back to the slave processors. In general, the computation time of the master processor is greater than that of any other processor.

Figure 3 Detailed breakdown of a two-part diff request (units are approximately 100 μs)



In the best case, the master processor arrives at the barrier first, so that the computation is not delayed by the receipt of barrier messages from the slaves. Thus, we model the time for a barrier as the sum of four components: (1) the time for the last processor to send a message to the master, (2) the time to merge write notices, (3) the time for the master to send a message to all the slaves, and (4) the average post-barrier computation time.

Networking overheads. In order to make it possible to change network parameters, we also break down the cost of sending and receiving messages. For this purpose, we used simple User Datagram Protocol/Internet Protocol (UDP/IP) benchmarks, but since the actual separation between hardware and software costs is difficult to determine without detailed system information, we rely on prior studies[16] to choose appropriate values for hardware latencies.

We divide send-side communication as follows: (1) software overhead to transfer a message to the kernel, to go down the protocol stack, and to set up the network adaptor to transfer the message, and (2) latency for the network controller to begin sending the message (assuming a direct memory access, or DMA, model). Costs for the receiver are measured similarly, but we have found in practice that they are close to those of the sender, and so we do not distinguish the two.

More importantly, the time to make a send or receive system call varies considerably from one invocation to the next, presumably because of caching or buffer allocation issues. For example, if we measure the cost of repeatedly sending a one-kilobyte message to a remote processor, in one case flushing the cache between system calls and in the other not, then the send ( ) system call time is 527 μs in the first case and 91 μs in the second for the Fast Ethernet on our system. As a result, we use a microbenchmark suite[17] to first estimate the basic hardware and software cost breakdown of a messaging operation, and then use the actual send ( ) and receive ( ) measurements from each application to adjust for the cache and operating system effects just mentioned.

Wire time for UDP/IP messages over Fast Ethernet is computed as message size plus protocol overheads (46 bytes) divided by the bandwidth; for UDP/IP messages over OC-3-based ATM, we use the established effective bit rate of 135 Mbps.[18]

The parameters for our two networks are shown in Table 2. Although the software protocol times are

Table 2 Network cost parameters for each of the two networks

Parameter	Fast Ethernet		OC3-Based ATM
	sz <= 1024	sz > 1024	All Sizes
Software protocol time (μs) (top is system call only, bottom is total)	71 + 0.03138 · sz 102 + 0.03865 · sz	71 + 0.02583 · sz 154 + 0.02583 · sz	74 + 0.01953 · sz 150 + 0.02918 · sz
Wire time (μs)	0.0800 · sz	0.0800 · sz	0.05926 · sz
Network hardware latency (μs)	25	25	50 (interface card) 10 (ATM switch)
Network protocol overheads (bytes)	58	58	40

Table 3 Analysis of empty time test application (all times in μs)

Component	Fast Ethernet (Per Fault)	ATM (Per Fault)
Total software protocol time	469	840
Extra send/receive system call time	747	597
Wire time	338	248
Network hardware latency time	100	220
Signal overhead	40	40
Remote handler compute time	283	236
Remote handler delay (from arrival to processing)	75	77
Total estimated time (and percent of total measured time)	2052 (96%)	2258 (96%)

generally linear in the size of the message, deviations of up to 40 percent can occur across message size boundaries that are powers of two. For example, the protocol times for 1024-byte and 1025-byte messages are 142.5 μs and 110.5 μs, respectively. It is impossible to deduce the sources of these deviations without detailed information from the vendors, and so, for simplicity, we chose to ignore these fluctuations, averaging out the results in the ranges shown.

Model validation for simple tests. Even though we collect a large amount of data for each application run, these are in the form of averages (e.g., average message size, average number of diff requests per fault, average computation times). Combined with our simple network models, the use of such averages will introduce some error in our estimates of execution time. In this subsection, we explore the accuracy of our model for faults and locks using the test applications provided by TreadMarks.

For each test that follows, including the ones described in the next section, we ran each application five times on a quiescent system and obtained both the means and variances for all measured parameters. In every case, we found that variances were quite small, with all measurements of a particular parameter differing by only a few percent.

Empty time. In the first test, we examine the cost of empty page faults, as would occur for cold page misses. The test program uses two processors, the second of which faults on 1024 pages managed by the first. Requests for page faults are 12 bytes in size; responses are 4 KB in size (corresponding to the page size). Signals have been measured on our system to take 40 μs. The results are shown in Table 3.

In the last row of the table, we show the estimated time for the operation according to our model, and relate this time to the total amount of time measured in the application for empty faults. (We do not show the actual measured time since it would be redundant.) Clearly, the cost of empty page faults is dominated by software protocol handling, which represents 60 percent and 63 percent of the total time for Fast Ethernet and ATM, respectively. As can be seen, the accuracy of the model is quite high at 96 percent for both tests.

Fault time. In our second test, we examine the cost of making diff requests. The test program has three phases. In the first phase, processor 0 obtains small 20-byte diffs from all other processors; in the second, all processors but the first exchange 20-byte diffs among themselves; and in the third phase, processor 0 obtains large 4120-byte diffs from all other processors. In the following discussion, the source refers to the processor requesting diffs and the destination(s) to the processor(s) from which diffs are being requested. The breakdown for each of

Table 4 Analysis of fault time test application (four processors)

Component	Fast Ethernet (Per Fault)			ATM (Per Fault)		
	Phase I	Phase II	Phase III	Phase I	Phase II	Phase III
Total software protocol time (src)	104 · 3 + 105	104 · 2 + 105	104 + 262 · 3	150 + 151 · 3	150 · 2 + 151	150 + 270 · 3
Extra send/receive syscall time (src)	42 · 3 + 0	157 · 2 + 0	57 + 154 · 3	0 + 0	121 · 2 + 0	58 + 89 · 3
Wire time	12	12	340	7	7	250
Network hardware latency time	100	100	100	220	220	220
Fault handler compute time	305	300	835	300	287	836
Signal overhead	40	40	40	40	40	40
Remote handler compute time	65	83	86	65	87	64
Total software protocol time (dest)	209	209	366	301	301	420
Extra send/receive syscall time (dest)	169	58	68	140	56	57
Remote handler delay (from arrival to processing)	262	455	368	267	354	370
Max signal-handling time at dest processors (for last four items)	1051	1331	2008	917	1179	1595
Total estimated time (and percent of total measured time)	1705 (97%)	1884 (87%)	3612 (85%)	1943 (94%)	2045 (88%)	3542 (83%)
Total estimated time using max time at dest (and percent of total measured time)	1759 (100%)	2189 (101%)	3900 (91%)	1785 (87%)	2155 (93%)	3508 (82%)

these phases for a four-processor experiment is shown in Table 4.

As described earlier, a multipart diff request is a pipelined operation where either the sending, receiving, or wire time limits the performance. This operation is shown in the table as an entry multiplied by the factor corresponding to number of processors involved in the multipart diff request. (The times for sending and receiving are broken down in the first two rows to allow the appropriate maximum value to be chosen.) Once again, the software protocol handling time represents a significant fraction of the total time, but DVSM software handling time and remote delays caused by message-handling contention can also be important.

We also show in Table 4 the maximum signal-handling time, which includes send or receive system call time, handler compute time, and delay in processing, at each of the destination processors, taking into account the times at which each request was sent. When the maximum time is large, it indicates that there is significant imbalance among the destination processors; in this case, using this value in the calculation can lead to more accurate results. Showing this value only serves to point out that imbalance exists; our model in the next section does not use this value because of the complexity of trying to do so.

Table 5 shows similar results for an eight-processor experiment.

Lock acquisition. We now examine the cost breakdown for lock acquisition. In the first test, two processors interact in such a way that the first makes repeated lock requests to the second. In the second test, three processors interact in such a way that the first makes repeated requests for locks managed by the second, but most recently accessed by the third. In all cases, the message sizes were 24 bytes for the requests, and four bytes for the replies. The results are shown in Table 6.

Discussion of errors. The previous section compared the model prediction of execution time components to measured values for simple test applications provided by TreadMarks. The accuracy of these models is affected by numerous factors, notably:

- Our model of the network is only approximate, as it assumes that costs grow linearly with the packet size, with no unusual variations; also, in the case of Fast Ethernet, we do not model the effects of packet collisions, which sometimes arise with larger numbers of nodes.
- Measurements of the delay at remote processes do not include delays arising from the remote process running in the kernel; in particular, if two

Table 5 Analysis of fault time test application (eight processors)

Component	Fast Ethernet (Per Fault)			ATM (Per Fault)		
	Phase I	Phase II	Phase III	Phase I	Phase II	Phase III
Total software protocol time (src)	104 · 7 + 105	104 · 6 + 105	104 + 262 · 7	150 + 151 · 7	150 · 6 + 151	150 + 270 · 7
Extra send/receive syscall time (src)	33 · 7 + 0	106 · 6 + 0	40 + 147 · 7	0 + 0	68 · 6 + 4	0 + 0
Wire time	12	12	340	7	7	250
Network hardware latency time	100	100	100	220	220	220
Fault handler compute time	700	984	1792	680	947	2154
Signal overhead	40	40	40	40	40	40
Remote handler compute time	68	107	94	66	97	66
Total software protocol time (dest)	209	209	366	301	301	420
Extra send/receive syscall time (dest)	187	101	107	138	60	60
Remote handler delay (from arrival to processing)	484	902	795	280	801	382
Max signal-handling time at dest processors (for last four items)	1492	3080	3526	1245	2687	2756
Total estimated time (and percent of total measured time)	2759 (98%)	3820 (70%)	6641 (77%)	2939 (97%)	3936 (75%)	5632 (73%)
Total estimated time using max time at dest (and percent of total measured time)	2586 (92%)	5581 (102%)	6351 (73%)	2493 (81%)	4270 (81%)	5840 (76%)

Table 6 Analysis of lock time test application

Component	Fast Ethernet (Per Fault)		ATM (Per Fault)	
	2 Processors	3 Processors	2 Processors	3 Processors
Total software protocol time	411	617	602	903
Extra send/receive syscall time	96	153	41	66
Wire time	12	18	6	10
Network hardware latency time	100	150	220	330
Lock acquisition compute time	132	135	133	139
Signal overhead	40	80	40	80
Remote handler compute time	32	62	33	63
Remote handler delay (from arrival to processing)	94	195	98	199
Total estimated time (and percent of total measured time)	917 (103%)	1410 (98%)	1173 (97%)	1790 (92%)

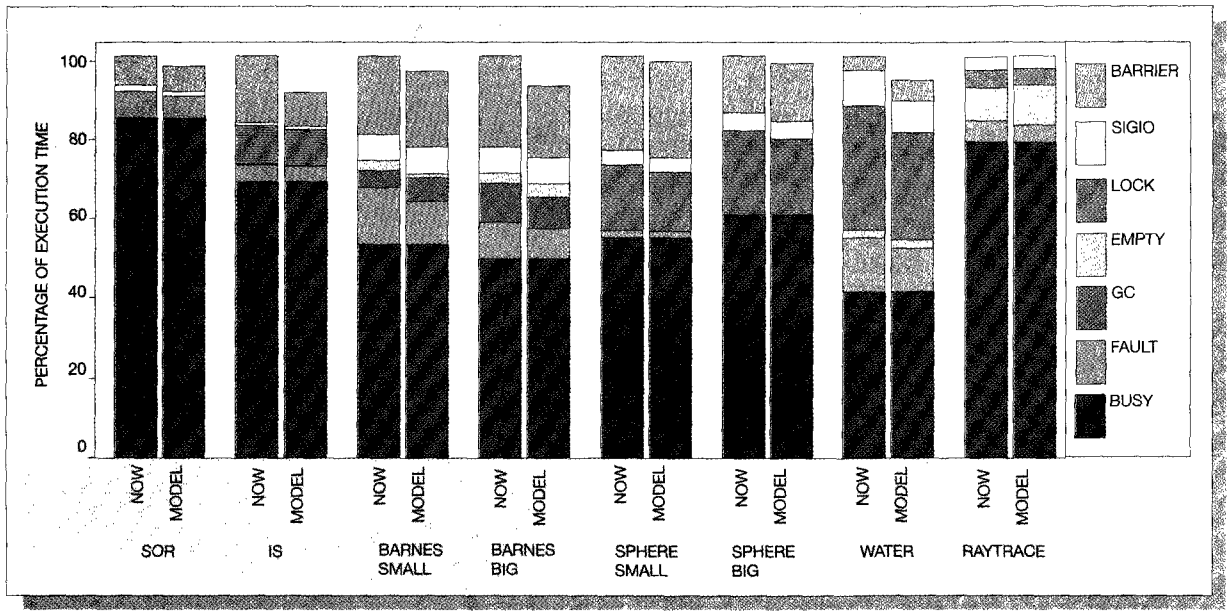
packets arrive nearly simultaneously, the delay on the second imposed by the interrupt-level processing of the first will not be captured.

- Finally, all our measurements only represent averages; it is particularly a problem with operations that involve multiple remote processors, in that it assumes that there is no imbalance, which is not usually the case (even if operations at the remote processors are identical).

Despite these sources of errors, our model correlates well with actual measurements. The only significant exception is for the case of diff requests involving many parts, primarily resulting from our optimistic model of interprocessor interactions. Since diff requests in practice have few parts, however, we do not expect these errors to be significant.

Model validation for full applications. The test applications demonstrate the basic accuracy of the model by performing exactly the same DVSM operation a large number of times. Real applications do not possess this uniformity and may be more greatly affected by the averaging that we perform. In par-

Figure 4 A comparison of the execution time, subdivided into its components, predicted by the model and normalized to the actual execution time



ticular, responses to diff requests may vary greatly, both in size and in the time required to compute and apply the diff, from one request to the next and from one processor to the next. In order to assess these effects, we now examine the accuracy of the model for the full applications summarized in Table 1.

To this end, Figure 4 presents the same information as in Figure 1, with the addition of a breakdown corresponding to the model prediction. As can be seen, we still exceed 90 percent accuracy in the prediction of the total execution time for all applications. Some overheads, such as operating system overhead and network contention, are not part of the model, and therefore the model typically underestimates some of the overhead components. (The biggest discrepancy is in barrier time in IS, but in this case, we are dealing with a small absolute error of about 90 milliseconds.) Even so, the model accurately predicts the relative importance of the various execution time components.

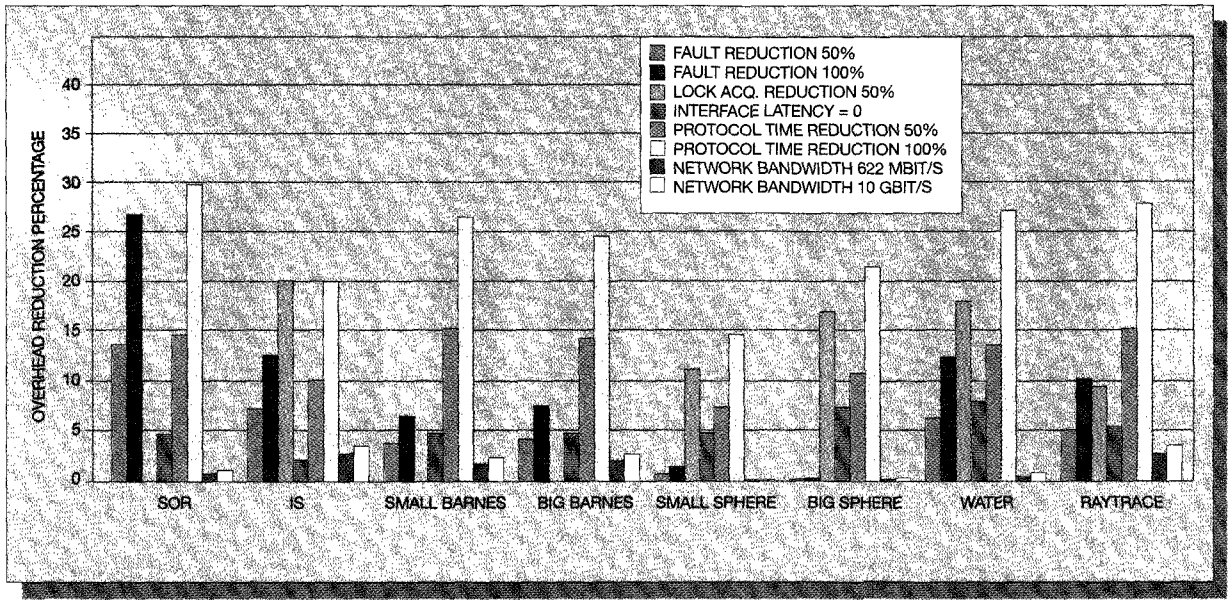
Performance trends anticipated for future technologies

In our model, we have carefully broken down the different costs associated with each significant DVSM operation. This breakdown now allows us to vary different technology parameters and estimate their effects on the execution time of each application. Next, we explore the effects of improvements in DVSM software, networking, and compiler technology on our current system. Then, we consider the effects of faster processors, given both current and future software, networking, and compiler technology. Finally, we consider as a case study a specific next-generation system.

Effects of DVSM improvements. There are two main causes for poor performance in applications running under TreadMarks: *lock acquisition* and *memory consistency* (which consists of empty, fault, and garbage collection times). The relative magnitude of these overheads, however, is highly dependent on the relative changes in different technology parameters. Figure 5 shows the overhead reduction when some of the more important parameters are changed. These parameters are (1) the number of faults, (2) lock acquisition costs, (3) protocol costs, and (4) network bandwidth and interface latency, each of which is described in detail in the following subsections.

Fault reduction. The fault-handling time is, of course, most effectively reduced if the application can be re-

Figure 5 Overhead reduction for changes in some technology parameters for current generation system



structured to minimize its need for communication. In the context of this study, however, we consider prefetching as a latency-hiding technique to overlap communication with computation, an approach that has already been shown to be effective in both shared-memory multiprocessors[19,20] and in networks of workstations.[12]

Although prefetching may reduce the latency associated with faults, it does not reduce the overhead due to requesting, computing, and applying diffs, and will likely increase traffic on the interconnection network as a result of mispredicted prefetches. In our model, the cost of a prefetch is the sum of software costs that are normally incurred at the source processor for a fault, but without the penalty of waiting for replies. (For the purpose of this analysis, we assume that no faults are mispredicted to show an upper bound on performance improvements. It is relatively straightforward to augment the model to include mispredicted prefetches.)

Applications whose overhead is dominated by fault-handling time benefit the most from fault time reduction. SOR is one such application in which the overhead times can be reduced by 13 percent if half of the faults can be prefetched (see Figure 5). However, the overhead is already relatively small for this application, and the resulting absolute performance improvement is limited.

Other applications benefit less since the relative importance of the fault-handling overhead time is smaller compared to SOR. Applications that have large-sized diffs, such as Barnes (over 3-Kbyte diffs on average), cannot fully benefit from prefetching, since the software costs at the source and destinations are quite high; moreover, when several diffs must be prefetched, all sends contribute to overhead in the source processor, rather than just the first for a regular fault. (The remaining sends occur when the processor would otherwise be idle waiting for the response.)

The only application that is almost unaffected by fault reduction is Sphere. The reason is that there is very little read and write sharing in Sphere; most data structures are initialised at the beginning of the execution and then subsequently only read. Thus far, we have only been considering dynamic prefetching based on reference history, but if explicit prefetch operations could be inserted based on compiler analysis, it would be possible for faults due to cold misses (i.e., empty time) to also be reduced.

Lock acquisition time reduction. Prefetching can also be used to hide the latency to acquire locks (by initiating the lock request somewhat before it is needed). Although locks are used by release consistency protocols to maintain consistency, we find that there is little contention for locks in the applications we studied, so the potential increase in lock holding time would not inflate contention unacceptably. It has previously been shown that lock prefetching requests can be generated automatically by a compiler algorithm.[13]

We include the effect of lock prefetching in the model in a manner similar to the case of data prefetching; the major difference in characteristics is that locks require a single send operation from the source and incur much smaller software overheads than faults. As shown in Figure 5, a 50 percent reduction in lock acquisition time translates to reductions in overhead ranging from 8 percent to 20 percent for IS, Sphere, Water, and Raytrace (i.e., all applications having locks).

Protocol reduction factor. The protocol reduction factor reduces all overheads in the model that are related to protocol execution. We find that applications with a high degree of locking overhead or memory consistency overhead (e.g., Barnes and Water), benefit greatly (15 percent reduction in overhead for a 50 percent protocol cost) from lower-latency communication facilities (relative to UDP/IP).

We believe that there is potential to reduce the protocol overhead significantly in future systems. This reduction can be achieved by using lighter-weight protocols or by simply using lower levels in the protocol stack (e.g., the AAL, or ATM adaptation layer, in the ATM interface). Furthermore, newer ATM interfaces may perform some of the protocol functions in hardware which would also reduce latencies.[21]

Network interface latency and other parameters. Finally, removing the network interface latency has a moderate effect on performance across all applications (except Water). The reason is, of course, that with current processor technology, the majority of the costs lie within software.

Other parameters that we studied did not have an appreciable effect on the execution time of applications. In particular, changes in the network bandwidth do not appear to be important, given our current processor speeds.

Effects of faster processors. Next, we consider the impact of processor performance, both on the efficiency of applications and on the benefit of the types of technology changes described in the previous section. As mentioned earlier, the performance of processors is increasing by a factor of two every 18 months, so in five years, processors may be eight times faster than the ones used today.

To model faster processors, we decrease the time spent in various components according to the increase in processing speed; the only components that are not modified are the network interface latency and the wire delay (arising from bandwidth limitations). In general, however, the benefits of faster processors cannot be fully realized in system software, because such software tends to have poor cache locality for both data and instructions.[22,23] For example, the extra system call time that we have included in our model arises because protocol code and data are often not found in cache. As another example, the minimum remote lock acquisition time reported by Cox et al. on 40-MHz processors[24] is actually lower than on our system. As a result, we consider two cases: one where all software gets the full benefit of faster processors, and another where the system software gets only a 50 percent benefit. (In particular, we model a system software benefit of x by increasing the processor speed by a factor of x every time the actual processor speed doubles, i.e., increases by 100 percent.) Although this value is only a rough estimate of what may occur in practice, it illustrates how such behaviour can affect performance.

Figure 6 shows the model's prediction of the efficiency of each of our applications as processor performance increases. We show the range of relative processor speeds, from 0.25 (roughly 1992) to 16.0 (roughly 2000). As expected, the efficiency decreases with increasing processor speed because hardware delays will become relatively more important. Some applications are affected less than others by this. Sphere, for instance, retains much of its efficiency because most of the overhead is in lock acquisition, which is highly software intensive in both DVSM and communication protocol code. For many of the applications, performance is relatively constant over a range of processor speeds if system software benefits fully from faster processors. If we consider the case where only 50 percent of the benefit can be achieved (Part B of Figure 6), efficiency varies more dramatically; in this case, all applications exhibit rapidly decreasing efficiency, with Barnes and Sphere being most severely affected.

Figure 6 The efficiency as a function of the processor performance (1 represents an IBM 43P-133 MHz)

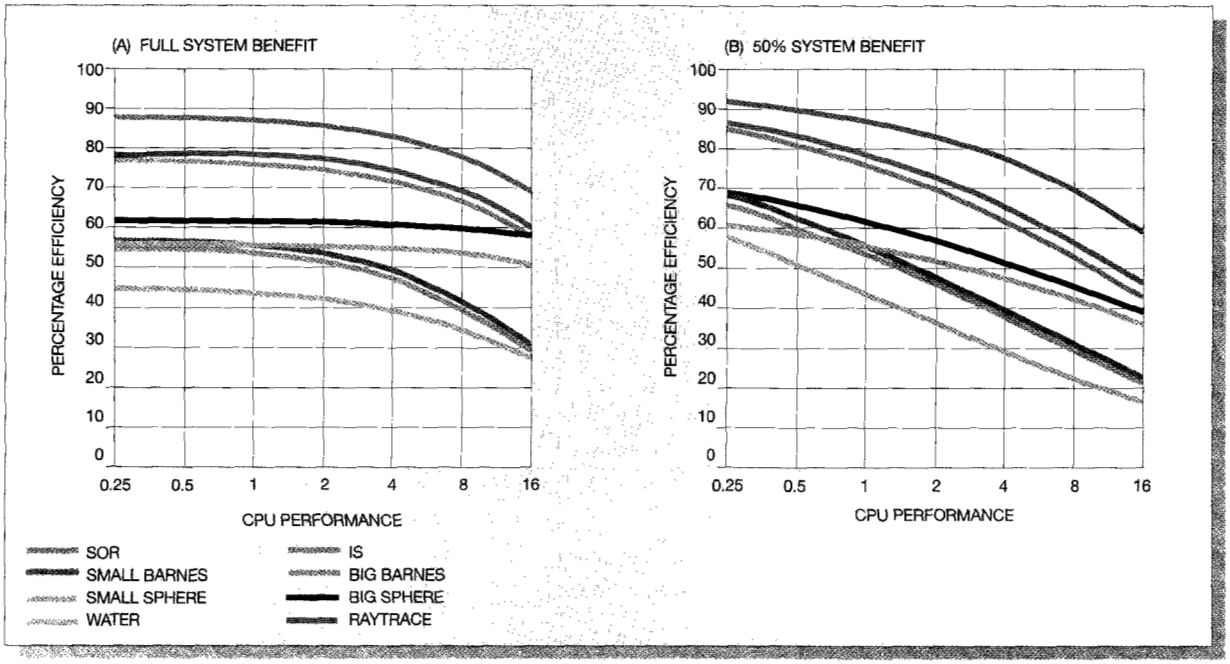


Table 7 Parameters of a future-generation system

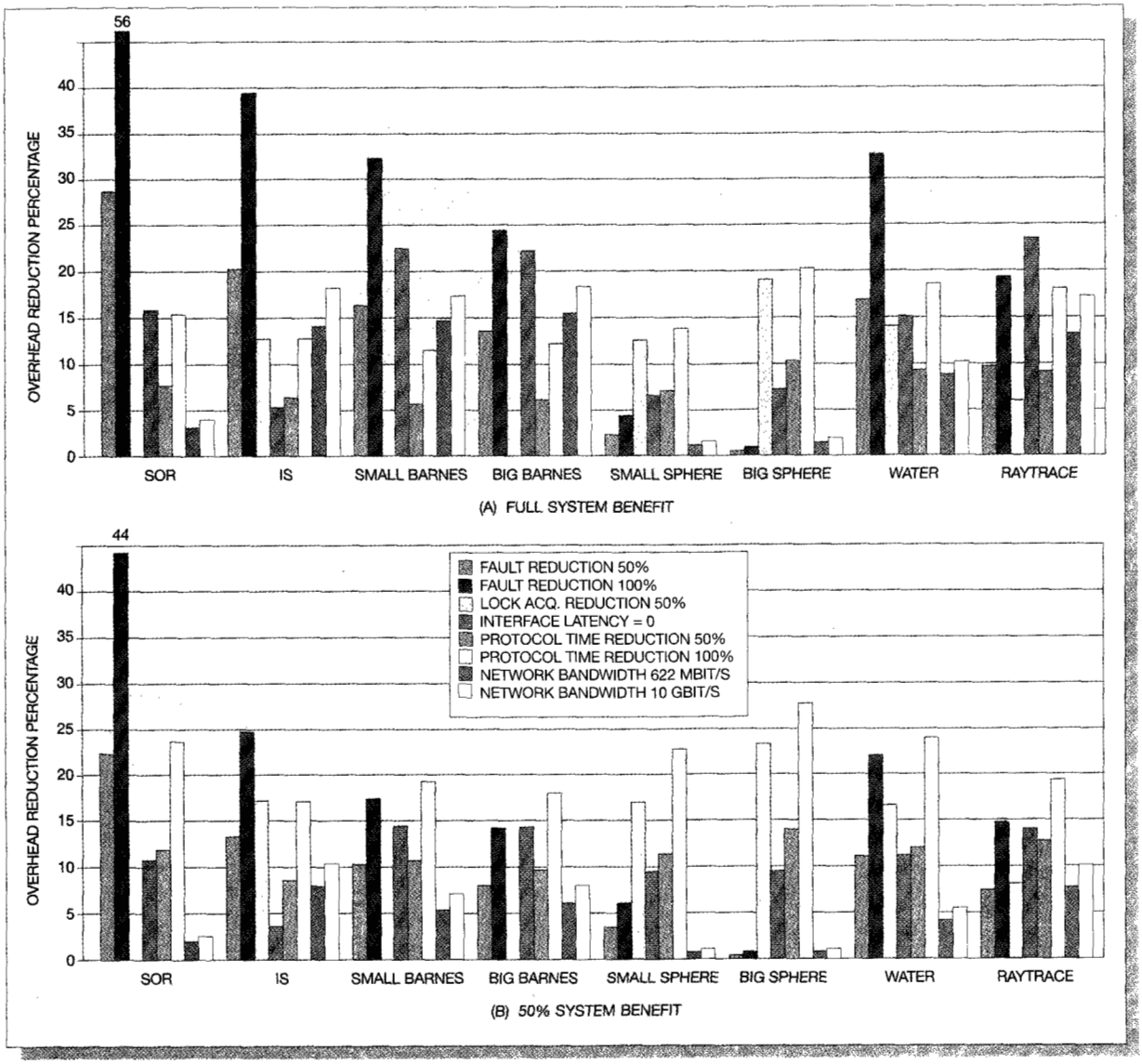
Processor performance factor	8
Network interface time	500 ns
Network bandwidth	10 Gbps
Signal latency time	625 ns
Network switch latency	500 ns
Protocol reduction factor	0.2
Fault reduction factor	0.5
Lock acquisition reduction factor	0.5

In Figure 7, we show the effects of the same overhead reductions considered in the previous section (and Figure 5), but this time using processors eight times more powerful. In this case, using prefetching for data appears to have the greatest impact on performance, ranging from a 15 percent to a 56 percent reduction in overhead for SOR, IS, Barnes, and Water. If system software cannot benefit fully from increases in processor speeds, however, then the benefit of prefetching will be reduced. Finally, factors that were insignificant with the slower processor, namely network interface latency and network bandwidth, are now much more important.

A representative future system. In five years time, processor performance will be about eight times that of today's high-performance microprocessors. If other technology parameters stay the same, we can see from Figure 6 that the processor efficiency will decrease. In order to anticipate the future situation, we have applied the model with a set of technology parameters that we believe will be realized within five years (or possibly sooner). These parameters are summarized in Table 7. Current VLSI (very large-scale integration) technology is sufficiently advanced to accommodate 10 Gbps networks and interfaces.[21] The signal latency time is likely to decrease with faster processors, and operating systems are likely to provide mechanisms for more effectively handling remote procedure calls.[16]

Part A of Figure 8 compares the breakdown of execution time of this future system with that of current technology, assuming system software can fully benefit from improvements in processor performance. As can be seen, it is not only possible to maintain the same level of efficiency as today, but also to actually improve it in all cases. The largest improvements are in memory consistency and lock acquisition operations, both overheads that are substantially

Figure 7 Overhead reduction for technology parameter changes when the processor performance is eight times current technology



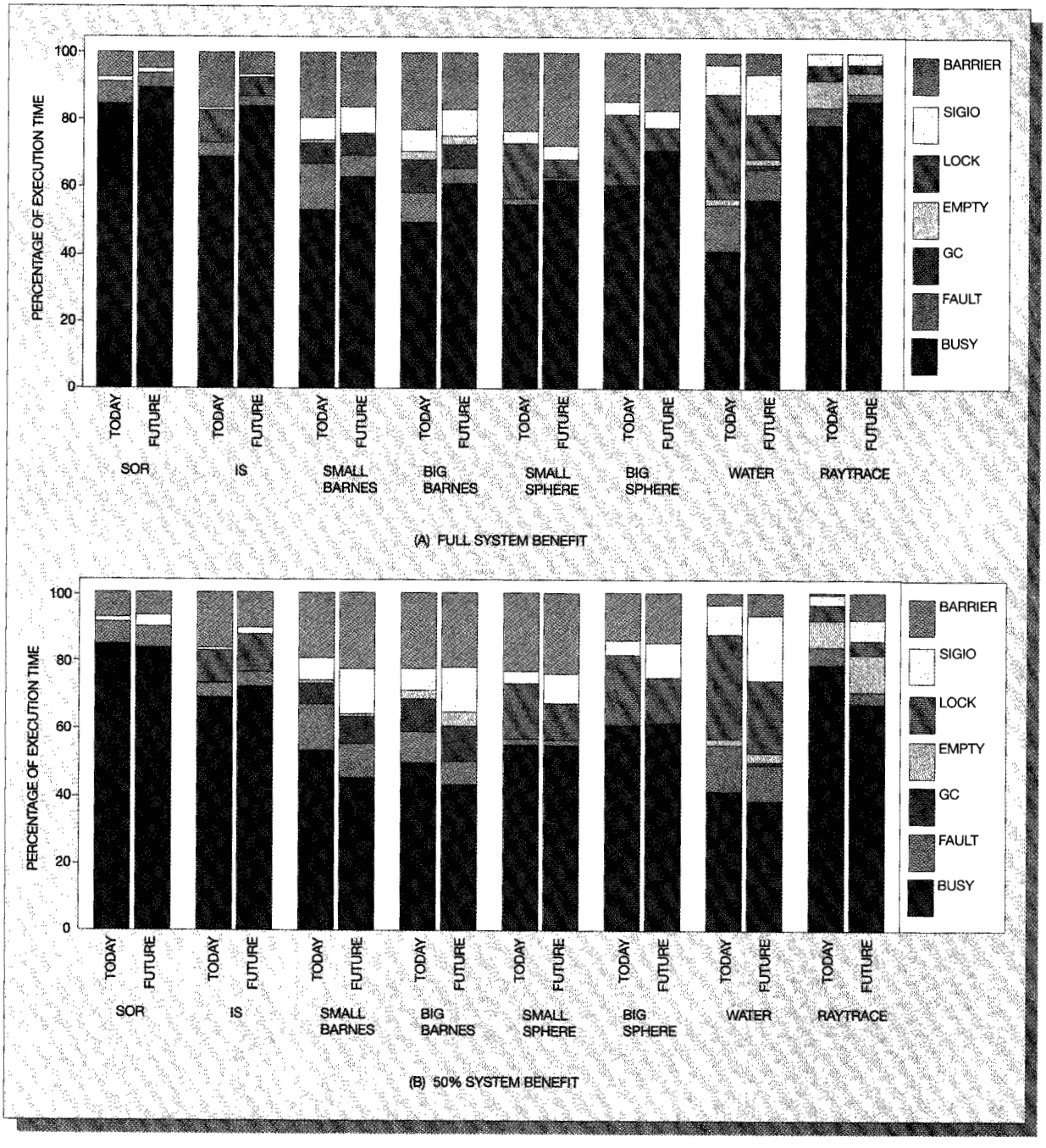
diminished from the reduction in protocol software costs.

Part B of Figure 8 shows the same comparison for the case where system software can only partially benefit from improvements in processor performance. In this case, the efficiency of current systems will not be sustained for most applications, even with the aggressive latency-hiding techniques that might be used. This reemphasises the fact that the single most important system component to optimize, even in future systems, is the communication protocol software.

Related work

There are many studies of the performance of various DVSM systems, but only a few of them present

Figure 8 Execution time breakdown for a future generation system compared to today's technology



models that can be used to predict performance of future systems.

Karlsson and Stenström[25] studied TreadMarks-based applications running on a system consisting

of a number of small-scale shared-memory multiprocessors connected by a standard OC-3 ATM switch. They used program-driven simulation to study the performance effects of faster processor, network, and interface technologies. Their study confirms our findings that the interface latency is the performance bottleneck, but in contrast to our study, and due to the particulars of their simulation model, they cannot separate communication software cost from the communication hardware overhead in the network interface. Also, because of the limited resources available when simulating a multiprocessor system, they are unable to run realistically sized workloads.

Dwarkadas et al.[26] studied the performance of different release consistency options for four applications, using predictions for the cost of various DVSM overheads. In their study, they examined the effects of varying software overheads and of higher-bandwidth networks on the performance of applications; they show how performance for sharing-intensive applications is most significantly affected by the software overheads. Later, Cox et al.[24] used a similar simulation approach (except using better estimates of software overheads) to study the performance of several DVSM applications in (what was then) a next-generation system; in this study, they also considered the effects of reductions in software overheads. Since these studies also used simulation, they used small data sets in their experiments.

Our work differs from these in that we (1) analyse a wider range of applications and larger problem sizes (since we could run on real hardware), (2) present detailed cost breakdowns for several DVSM operations, showing the high software costs actually incurred in modern systems, (3) develop a model of DVSM applications, based on these breakdowns, and (4) use this model in conjunction with present-day measurements to predict the effect that a variety of technological changes may have on the performance of these applications. Most significant is that we present a way to analytically model the performance of applications.

Apart from TreadMarks many other DVSM systems have been proposed and evaluated in the literature. We have chosen to mention a few of the more relevant ones:

- SoftFLASH[27] implements a clustered system similar to the one studied by Karlsson et al.[25] A major difference from TreadMarks is that SoftFLASH is implemented at kernel rather than at user level.
- CVM[28] is a new DVSM system based on the TreadMarks experience. It is designed in C++ with support for multiple consistency models. In his study, Keleher argues that it is actually not necessary to support multiple writers to a shared page, but rather that lazy release consistency is most important.
- Finally, Blizzard-S,[29] Shasta,[30] and Aurora[31] represent DVSM systems that are based on shared objects rather than pages, as in the case of the previously mentioned systems. Since they cannot rely on hardware mechanisms to detect shared-memory accesses, they modify the executable code to check the state of shared objects before accessing them and to invoke the protocol software if needed. None of these studies, however, examines the performance effect of future processors and communications technology.

The model technique developed and used in this paper can also be applied to all the above-mentioned systems in order to study performance effects of changing technologies. Software probes need to be inserted into the protocol software. This is probably very easy in the systems that run entirely in user mode, whereas in the case of SoftFLASH, access to the kernel would be required. The actual model in this paper is, of course, targeted to TreadMarks and would therefore need to be changed to suit the different protocols accordingly.

Conclusions

A network of workstations is an attractive platform for parallel computing because of its potential to deliver high performance at a relatively low cost. It has previously been shown that it is even possible to achieve reasonable performance for a shared-memory programming model if the consistency model is relaxed.[5]

Given that processor performance has been increasing much more rapidly than network performance since the earlier measurements of DVSM performance were done,[5] it is expected that application efficiency would be lower for the same problem size on current systems, as we have observed. In this paper, we found that a distributed virtual shared-memory system on a network of workstations indeed can still deliver cost-effective performance, even when using present-day commodity network technology. It is, however, limited to a class of applications that has a sufficiently high computation-communication ratio, such as those examined in this paper.

Our work shows how a model can be developed for parallel applications running on a DVSM system, which we use to study their performance as changes occur in technology. As has been noted before, the main bottleneck for DVSM systems with current technology is the software overhead in the communication protocol. Latency-hiding techniques to reduce fault-handling time and lock acquisition time can be effective if implemented. However, with the expected performance of future processors, the performance of these applications is likely to be constrained more by hardware-related delays such as network interface and wire time.

Our model shows that the anticipated improvements in DVSM and networking technology are likely to permit the same relative application performance to be maintained over the near to medium term. However, if system software cannot be written so as to take full advantage of faster processors, it will be almost impossible to achieve the same speedups (equivalent efficiency) as today. Also, it will not be possible to maintain this performance if latency-hiding techniques are not used for memory faults and lock acquisitions.

In this study, we have focussed on how changes in protocol and DVSM software and in processor and network hardware will affect speedup for a set of applications of specific size executing on eight processors. Problems of greatest importance that will execute on DVSM systems of the future will involve much larger computations (e.g., sequential execution times of hours or days rather than minutes or seconds) and will require many more processors. The modelling approach described in this paper can be immediately applied to any problem by running it on an existing system to gather the base statistics. With a deeper understanding of the various application parameters that are used by the model, it would also be possible to apply the approach to problems too large to be run today.

Acknowledgments

The network of workstations used for this study is part of the Parallelism on Workstations (POW) project, which is a cooperative project between the University of Toronto and the Centre for Advanced Studies at the IBM Toronto Development Laboratory. Three major themes within the POW project are (1) development of compilers that support automatic parallelization for a network of workstation environments, (2) exploitation of prefetching to overcome remote data access latencies in distributed-memory systems, and (3) efficient multiprogrammed scheduling of workloads dominated by parallel jobs.

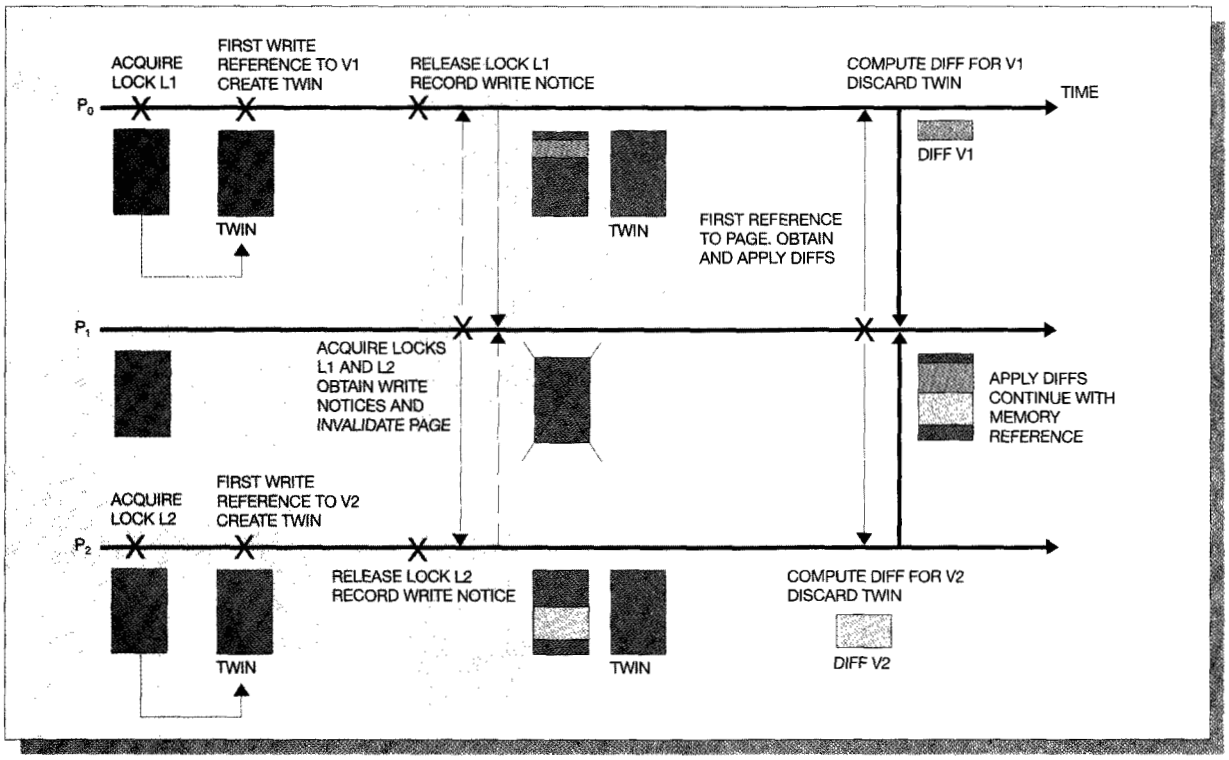
We would like to thank Giridhar Chukkapalli who kindly provided the Sphere application. The research in this paper was supported in part by the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of Ontario, Northern Telecom, and the Swedish National Board for Industrial and Technical Development (NUTEK) under project number P855.

Appendix

To illustrate the LRC protocol, consider three processes sharing a single distributed virtual shared-memory page that contains two variables, V1 and V2, each protected by a lock, L1 and L2, respectively. Figure 9 depicts a sequence of actions taken by the processors. Initially, the page is marked as valid, but write-protected in all three processors; all processors can read the variables. Next, processor 0 acquires the lock L1, and roughly at the same time processor 2 acquires lock L2. When these processors modify the variables corresponding to the lock they acquired for the first time, a page fault will occur (since the page was initially write-protected), and a local copy of the page will be made in each processor; these copies, or *twins*, can later be used to determine which portions of the page have been modified. The page is then unprotected, allowing reads and writes to proceed uninterrupted. Later, when the processors release their locks, the fact that the page has been modified is recorded in a *write notice*.

After the locks have been released, processor 1 acquires both locks, presumably to either read or write variables V1 and V2. Acquiring a lock involves sending a message to a preassigned manager of the lock, which forwards the request to the processor that last held the lock, which in turn responds with any write notices that are associated with the lock. In our example, a message is sent to both processors 0 and 2, both of which respond with a write notice for the same page, causing processor 1's copy of the page to be invalidated. When processor 1 subsequently accesses the page, a request is made to both other processors for *diffs*, which record what changed in the page on a given processor. (A diff is computed by comparing the current copy of the page against its twin.) After the diffs have been computed, the twins can be safely discarded.

Figure 9 The lazy release consistency protocol in TreadMarks. (Processors are not notified of changes until the lock acquisition, and pages do not get updated until the first reference after the lock acquisition.)



Processor 1 uses the diffs it receives to update its own copy of the page with the modifications made by processors 0 and 2. Hence, once processor 1 has received and applied both diffs, there will be three different versions of the page: one each on processors 0 and 2 that reflect the changes done to the page locally and one on processor 1 with an updated status containing changes made by both processors 0 and 2. In order for this multiple-writer scheme to work, it is assumed that the programmer does not associate overlapping memory regions with different locks, since that would cause the diffs to partly relate to the same addresses, and the final state of a shared page would depend on the order in which the diffs were applied.

TreadMarks also supports *barrier synchronisations* which, in addition to synchronising all processors, also cause the processors to exchange write notices for all shared-memory pages. Each barrier is associated with a managing processor that coordinates the actions of other processors. Basically, the manager collects the write notices from all other processors as they reach the barrier, and then redistributes them back to all processors involved in the computation. TreadMarks uses barriers to initiate garbage collection (GC) if the amount of memory consumed by write notices, diffs, and twins exceeds a predefined threshold on any processor. If garbage collection is initiated, then at the end of the barrier, all processors compute and exchange diffs for all pages, making all copies of each shared-memory page identical.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of PARALLELTOOLS, LLC, Digital Equipment Corporation, 3Com, Inc., or X/Open Company, Ltd.

Cited references

1. J. P. Singh, A. Gupta, and M. Levoy, "Parallel Visualization Algorithms: Performance and Architectural Implications," *Computer* **27**, No. 7, 45–55 (July 1994).

2. A. H. Karp, K. Miura, and H. Simon, "1992 Gordon Bell Prize Winners," *Computer* **26**, No. 1, 77–82 (January 1993).
3. A. H. Karp, M. Heath, D. Heller, and H. Simon, "1994 Gordon Bell Prize Winners," *Computer* **28**, No. 1, 68–74 (January 1995).
4. A. H. Karp, M. Heath, and A. Geist, "1995 Gordon Bell Prize Winners," *Computer* **27**, No. 1, 79–85 (January 1996).
5. C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *Computer* **29**, No. 2, 18–28 (February 1996).
6. P. Keleher, *Lazy Release Consistency for Distributed Shared Memory*, Ph.D. thesis, Department of Computer Science, Rice University, Houston, TX (January 1995).
7. P. Keleher, A. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," *Proceedings of the 19th International Symposium on Computer Architecture* (May 1992), pp. 13–21.
8. R. Mraz, D. Freimuth, E. Nowicki, and G. Silberman, "Using Commodity Networks for Distributed Computing Research," *Proceedings of Workshop on Computer Networking: Putting Theory to Practice, 1995 Asian Computing Science Conference*, Pathumthani, Thailand (December 1995), pp. 6–13.
9. E. Felten, R. Alpert, A. Bilas, M. Blumrich, D. Clark, S. Damianakis, C. Dubnicki, L. Iftode, and K. Li, "Early Experience with Message-Passing on the SHRIMP Multicomputer," *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, Philadelphia, PA (May 22–24, 1996), pp. 296–307.
10. T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (December 1995), pp. 40–53.
11. T. Mowry, A. Demke, and O. Krieger, "Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications," *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (1996), pp. 3–18.
12. M. Karlsson and P. Stenström, "Effectiveness of Dynamic Prefetching in Multiple-Writer Distributed Virtual Shared Memory Systems," to be published in *Journal of Parallel and Distributed Computing* (September 1997).
13. M. Karlsson and P. Stenström, "Lock Prefetching in Distributed Virtual Shared Memory Systems—Initial Results," *IEEE CS Technical Committee on Computer Architecture Newsletter*, 41–48 (March 1997).
14. G. Chukkapalli, personal communication, Department of Mechanical Engineering, University of Toronto, Toronto (e-mail: chuk@drill.me.utoronto.ca).
15. J. P. Singh, J. L. Hennessy, and A. Gupta, "Implications of Hierarchical N-body Methods for Multiprocessor Architecture," *ACM Transactions on Computer Systems* **13**, No. 2, 141–202 (May 1995).
16. C. A. Thekkath and H. M. Levy, "Limits to Low-Latency Communication on High-Speed Networks," *ACM Transactions on Computer Systems* **11**, No. 2, 179–203 (May 1993).
17. L. McVoy and C. Staelin, "Imbench: Portable Tools for Performance Analysis," *Proceedings of the USENIX 1996 Annual Technical Conference* (1996), pp. 279–294.
18. L. G. Cuthbert and J.-C. Sapanel, Chapter 2 in *ATM—The Broadband Telecommunications Solution*, The Institution of Electrical Engineers, London, UK (1993).
19. F. Dahlgren and P. Stenström, "Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems* 7, No. 4, 385–398 (April 1996).
20. T. Mowry and A. Gupta, "Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors," *Journal of Parallel and Distributed Computing* **12**, No. 2, 87–106 (June 1991).
21. P. Sundström, M. Karlsson, and P. Andersson, "An Interface Architecture for a Low-Latency Network of Workstations Using 10 Gbit/s Switched LAN Technology," *Proceedings of the IASTED International Conference on Parallel and Distributed Systems*, Euro-PDS'97, Barcelona (June 1997), pp. 166–176.
22. J. Chen and B. Bershad, "The Impact of Operating System Structure on Memory System Performance," *Proceedings of the Fourteenth Symposium on Operating System Principles* (1993), pp. 120–133.
23. A. Maynard, C. Donnelly, and B. Olszewski, "Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads," *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1994), pp. 145–156.
24. A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel, "Software Versus Hardware Shared-Memory Implementation: A Case Study," *Proceedings of the 21st International Symposium on Computer Architecture* (1994), pp. 106–117.
25. M. Karlsson and P. Stenström, "Performance Evaluation of a Cluster-Based Multiprocessor Built from ATM Switches and Bus-Based Multiprocessor Servers," *Proceedings of the 2nd Conference on High Performance Computer Architecture* (February 1996), pp. 4–13.
26. S. Dwarkadas, P. Keleher, A. Cox, and W. Zwaenepoel, "Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology," *Proceedings of the 20th Annual International Symposium on Computer Architecture* (1993), pp. 144–155.
27. A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy, "SoftFLASH: Analysing the Performance of Clustered Distributed Virtual Shared Memory," *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1996), pp. 210–221.
28. P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models," *Proceedings of the 16th International Conference on Distributed Computing Systems* (May 28, 1996), pp. 91–98.
29. I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-Grain Access Control for Distributed Shared Memory," *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)* (October 1994), pp. 297–307.
30. D. J. Scales, K. Gharachorloo, and C. A. Thekkath, "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory," *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (October, 1996).
31. P. Lu, "Aurora: Scoped Behaviour for Per-Context Optimized Distributed Data Sharing," *Proceedings of the 11th International Parallel Processing Symposium* (April 1997), pp. 467–473.

General references

J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Co-

herence," *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming* (1990), pp. 168–176.

K. Gharachorloo, A. Gupta, and J. L. Hennessy, "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1991), pp. 245–257.

L. Iftode, J. P. Singh, and K. Li, "Understanding Application Performance on Shared Virtual Memory Systems," *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, Philadelphia, PA (May 22–24, 1996), pp. 122–133.

K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," *Proceedings of 1988 International Conference on Parallel Processing* (1988), pp. 94–101.

J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News* **20**, No. 1, 5–44 (March 1992).

Accepted for publication May 20, 1997.

Eric W. Parsons *Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G4 (electronic mail: eparsons@cs.toronto.edu).* Dr. Parsons currently works at the Computing Technology Laboratory at Northern Telecom. He recently completed his Ph.D. in the Department of Computer Science at the University of Toronto in the area of multiprocessor scheduling. His primary research interests are in performance analysis, particularly in relation to multiprocessor systems and mobile computing. He will be joining the Department of Electrical and Computer Engineering at the University of Toronto as an assistant professor in 1998.

Mats Brorsson *Department of Information Technology, Lund University, P.O. Box 118, SE-221 00 Lund, Sweden (electronic mail: Mats.Brorsson@it.lth.se).* Dr. Brorsson is an associate professor in the Department of Information Technology at Lund University, Sweden. His main research interests are in parallel architectures and, in particular, performance analysis of shared-memory parallel applications. He received the M.Sc. and Ph.D. degrees in 1985 and 1994, respectively, both from Lund University.

Kenneth C. Sevcik *Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G4 (electronic mail: kcs@cs.toronto.edu).* Dr. Sevcik is a professor of computer science with a cross-appointment in electrical and computer engineering at the University of Toronto. He was the past Director of the Computer Systems Research Institute and past Chairman of the Department of Computer Science. He received a B.S. in mathematics from Stanford University in 1966 and a Ph.D. in information science from the University of Chicago in 1971. His primary area of research interest is in developing techniques and tools for performance evaluation, and applying them in such contexts as distributed systems, database systems, local area networks, and parallel computer architectures.

Reprint Order No. G321-5657.