Strings: A High-Performance Distributed Shared Memory for Symmetrical Multiprocessor Clusters*

Sumit Roy and Vipin Chaudhary †
Parallel and Distributed Computing Laboratory
Department of Electrical and Computer Engineering
Wayne State University
Detroit, Michigan 48202
{sroy,vchaud}@ece.eng.wayne.edu

†This research was supported in part by NSF grants MIP-9309489, EIA-9729828, US Army Contract DAEA 32-93D004 and Ford Motor Company grants 96-136R and 96-628R

Abstract

This paper introduces Strings, a high performance distributed shared memory system designed for clusters of symmetrical multiprocessors (SMPs). The distinguishing feature of this system is the use of a fully multi-threaded runtime system, written using POSIX threads. Strings also allows multiple application threads to be run on each node in a cluster. Since most modern UNIX systems can multiplex these threads on kernel level light weight processes, applications written using Strings can use all the processors in a SMP machine. This paper describes some of the architectural details of the system and analyzes the performance improvements with two example programs and a few benchmark programs from the SPLASH-2 suite.

1. Introduction

Though current microprocessors are getting faster at a very rapid rate, there are still some very large and complex problems that can only be solved by using multiple cooperating processors. These problems include the so-called *Grand Challenge Problems*, such as Fuel combustion, Ocean modeling, Image understanding, and Rational drug design. There has recently been a decline in the number of specialized parallel machines being built to solve such problems. Instead, many vendors of traditional workstations have adopted a design strategy wherein multiple state-of-the-art microprocessors are used to build high performance shared-memory parallel workstations. These symmetrical multiprocessors (SMPs) are then connected through high speed networks or switches to form a scalable computing cluster. Examples of this important class of machines include the SGI Power Challenge Array, the IBM SP2 with multiple PowerPC based nodes, the Convex Exemplar, the DEC AdvantageCluster 5000, the SUN HPC cluster with the SUN Cluster Channel, as well as the Cray/SGI Origin 2000 series.

Existing sequential application programs can be automatically converted to run on a single SMP node through the use of parallelizing compilers such as KAP [15], Parafrase [22], Polaris [20] and SUIF [1]. However, using multiple nodes requires the programmer to either write explicit message passing programs, using libraries like MPI or PVM; or to rewrite the code using a new language with parallel constructs eg. HPF and Fortran 90. Message passing programs are cumbersome to write and have to be tuned for each individual architecture to get the best possible performance. Parallel languages work well with code that has regular data access patterns. In both cases the programmer has to be intimately familiar with the application program as well as the target architecture. The shared memory model is easier to program since the programmer does not have to worry about the data layout and does not have to explicitly send data from one process to another. However, hardware shared memory machines do not scale that well and/or are very expensive to build. Hence, an alternate approach to using these computing clusters is to provide an illusion of logically shared memory over physically distributed memory, known as a Distributed Shared Memory (DSM) or Shared Virtual Memory (SVM). Recent research projects with DSMs have shown good performance, for example IVY [16], Mirage [10], Munin [3], TreadMarks [2], Quarks [14], and CVM [12]. This model has been shown to give good results for programs that have irregular data access patterns that cannot be analyzed at compile time [17], or indirect data accesses that are dependent on the input data-set.

DSMs share data at the relatively large granularity of a virtual memory page and can suffer from a phenomenon known as "false sharing", wherein two processes simultaneously attempt to write to different data items that reside on the same page. If only a single writer is permitted, the page may ping-pong between the nodes. A simple solution is to "hold" on to a freshly arrived page for some time before releasing it to another requester [10]. Relaxed memory consistency models that allow multiple concurrent writers have also been proposed to alleviate this symptom [4, 5, 2, 26]. These systems ensure that all nodes see the same data at well defined points in the program, usually at synchronization points. Extra effort is required to ensure program correctness in this case.

One technique that has been investigated recently to improve DSM performance is the use of multiple threads of control in the system. Multi-threaded DSMs have been described as third generation systems [23]. Published efforts have been restricted to non-preemptive, user-level thread implementations [14, 24]. Since the kernel does not know about user level threads, they cannot be scheduled across multiple processors on an SMP. Since SMP clusters are increasingly becoming the norm for High Performance Computing sites, we consider this to be an important problem to be solved. This paper introduces *Strings*, a multi-threaded DSM, based on Quarks. The distinguishing feature of *Strings* is that it is built using POSIX threads, which can be multiplexed on kernel light-weight processes. The kernel can schedule these lightweight processes across multiple processors for better performance. *Strings* is designed to exploit data parallelism at the application level and task parallelism at the runtime level. We show the impact of some of our design choices using some example programs as well as some benchmark programs from the SPLASH-2 suite [25]. Though similar work has been demonstrated with SoftFLASH [8], our implementation is completely in user space and thus more portable. Some other research has studied the effect of clustering in SMPs using simulations [11]. We have shown results from runs on an actual network of SMPs. Brazos [23] is another DSM system designed to run on multiprocessor cluster, but only under Windows NT. The *Strings* runtime has currently been ported to Solaris 2.6, Linux 2.1.64, and AIX 4.1.5.

The following section describes some details of the software system. The evaluation platform and programs for the performance analysis are described in section 3. Experimental results are shown and analyzed in section 4. Section 5 suggests some direction for future work and concludes the paper.

2. System details

The *Strings* distributed shared memory is based on the publicly available system Quarks [14]. We briefly describe the Quarks system and then explain the modifications carried out, and new features added to create *Strings*.

2.1. Execution model

The Quarks system consists of a library that is linked with a shared memory parallel program, and a *server* process running on a well known host. A program starts up and registers itself with the *server*. It then forks processes on remote machines using *rsh*. Each forked process in turn registers itself with the *server*, and then creates a *DSM_server* thread, which listens for requests from other nodes. The master process creates shared memory regions coordinated-ordinated by the *server* in the program initialization phase. The *server* maintains a list of region identifiers and global virtual addresses. Each process translates these global addresses to local addresses using a page table. Application threads are created by sending requests to the appropriate remote *DSM_servers*. Shared region identifiers and global synchronization primitives are sent as part of the thread create call. The newly created threads obtain the global addresses from the server and map them to local memory. The virtual memory sub-system is used to enforce proper access to the globally shared regions. The original Quarks system used user level *Cthreads* for implementing part of the system, but allowed only a single application thread. *Strings* allows multiple application threads to be created on a single node. This increases the concurrency level on each node in a SMP cluster.

2.2. Kernel level threads

Threads are light-weight processes that have minimal execution context, and share the global address space of the program. The time to switch from one thread to another is very small when compared to the context switch required for full-fledged processes. Moreover the implicit shared memory leads to a very simple programming model. Thread implementations are distinguished as being user-level, usually implemented as a library, or kernel level in terms of light-weight processes. Kernel level threads are a little more expensive to create, since the kernel is involved in managing them. However, user level threads suffer from some important limitations. Since they are implemented as a user level library, they cannot be scheduled by the kernel. If any thread issues a blocking system call, the kernel considers the process as a whole, and thus all the threads in it, to be blocked. Also, on a multiprocessor system, all user level threads can only run on one processor at a time. User level threads do allow the programmer to exercise very fine control on their scheduling within the process. Kernel level threads can be scheduled by the operating system across multiple processors. Most modern UNIX implementations provide a light-weight process interface on which these threads are then multiplexed. The thread package adopted for *Strings* was the standard Posix 1003.1c threads as available with Solaris 2.6. Since thread scheduling is handled by the kernel some parts of the system had to be changed to ensure correctness. Multi-threading has been suggested for improving the performance of scientific code by overlapping communications with computations [9]. Previous work on multi-threaded message passing systems has pointed out that kernel-level implementations show better results than user level threads for a message size greater than 4k bytes [21]. Since the page size is usually 4k or 8k bytes, it suggests that kernel threads should be useful for DSM systems.

2.3. Shared memory implementation

Shared memory in the system is implemented by using the UNIX *mmap* call to map a file to the bottom of the stack segment. Quarks used anonymous mappings of memory pages to addresses determined by the system, but this works only with statically linked binaries. With dynamically linked programs, it was found that due to the presence of shared libraries *mmap* would map the same page to different addresses in different processes. While an address translation table can be used to access opaquely shared data, it is not possible to pass pointers to shared memory this way, since they would potentially address different regions in different processes. An alternate approach would be to preallocate a very large number of pages, as done by CVM and TreadMarks, but this associates the same large overhead with every program, regardless of its actual requirements.

The *mprotect* call is used to control access to the shared memory region. When a thread faults while accessing a page, a page handler is invoked to fetch the contents from the owning node. *Strings* currently supports sequential consistency using an invalidate protocol, as well as release consistency using an update protocol [5, 14]. The release consistency model implemented in Quarks has been improved by aggregating multiple diffs to decrease the number of messages sent.

Allowing multiple application threads on a node leads to a peculiar problem with the DSM implementation. Once a page has been fetched from a remote node, its contents must be written to the corresponding memory region, so the protection has to be changed to *writable*. At this time no other thread should be able to access this page. User level threads can be scheduled to allow atomic updates to the region. However, suspending all kernel level threads can potentially lead to a deadlock, and would also reduce concurrency. The solution used in *Strings* is to map every shared region to two different addresses. It is then possible to write to the 'secondary region', without changing the protection of the primary memory region.

2.4. Polled network I/O

Early generation DSM systems used interrupt driven I/O to obtain pages, locks etc. from remote nodes. This can cause considerable disruption at the remote end, and previous research tried to overcome this by aggregating messages, reducing communication by combining synchronization with data, and other such techniques [19]. *Strings* uses a dedicated communication thread, which monitors the network port, thus eliminating the overhead of interrupt driven I/O. Incoming message queues are maintained for each active thread at a node, and message arrival is announced using condition variables. This prevents wasting CPU cycles with busy waits. A reliable messaging system is implemented on top of UDP.

2.5. Concurrent server

The original Quarks *DSM_server* thread was an iterative server that handled one incoming request at a time. It was found that under certain conditions, lock requests could give rise to a deadlock between two communicating processes. *Strings* solves this by forking separate threads to handle each incoming request for pages, lock acquires and barrier arrivals. Relatively fine grain locking of internal data structures is used to maintain a high level of concurrency while guaranteeing correctness when handling multiple requests.

2.6. Synchronization primitives

Quarks provides barriers and locks as shared memory primitives. *Strings* also implements condition variables for flag based synchronization. Barriers are managed by the master process, and every application thread sends an arrival message to the manager. Dirty pages are also purged at this time, as per Release Consistency Semantics [5].

Lock ownership is migratory with distributed queues. For multiple application threads, only one lock request is sent to the current owner, the subsequent ones are queued locally, as are incoming requests. The current implementation does not give any preference to local request, which guarantees progress and fairness, though other research has shown such optimizations to work well in practice [24].

3. Performance analysis

We evaluated the performance of *Strings* using programs from the SPLASH-2 benchmark suite [25]. These programs have been written for evaluating the performance of shared address-space multiprocessors and include application kernels as well as full fledged code. We show performance results for Fast Fourier Transform (FFT), LU decomposition with contiguous blocks (LU-c) as well as non-contiguous blocks (LU-n), the two versions of water ie. spatial (WATER-sp) and nsquared (WATER-n2) and radix sort (RADIX). Additionally we show results for matrix multiplication (MATMUL) and a real application, a program for deblurring images from Magnetic Resonance Imaging (MRI). The problem sizes are shown in Table [1]. All other parameters were left at their default value.

Program	Parameters
FFT	1048576 points
LU-c	2048 × 2048, block size 128
LU-n	512 × 512, block size 32
WATER-sp	4096 molecules
RADIX	1048576 integers
WATER-n2	4096 molecules
MRI	14 frequency points, PHANTOM image
MATMUL	1024 × 1024 doubles, 16 blocks

Table 1. Problem Sizes

3.1. Evaluation environment

Our experiments were carried out so as to show how various changes in the system impact performance. The runs were carried out on a network of four SUN UltraEnterprise Servers, connected using a 155 Mbs ForeRunnerLE 155 ATM switch. The first machine is a 6 processor UltraEnterprise 4000 with 1.5 Gb memory. The *server* process was always run on this machine. Two other machines are 4 processor UltraEnterprise 3000s, with 0.5 Gb memory each. These three machines all use 250 MHz UltraSparc processors. The last machine is also a 4 processor Enterprise 3000, with 0.5 Gb memory, but using 167 MHz UltraSparc processors. Since all runs were carried out on the same platform, we do not expect our results to be qualitatively affected by this slight load imbalance.

For each case, the following runs were carried out:

P16T1-KT: sixteen processes, four per machine, with a single application thread per process. The runtime uses kernel threads.
This is the base case and approximates the typical environment used in previous work on DSMs eg. TreadMarks has been studied on ATM networked DECstation-5000/240s [2], *CVM* results were presented on the IBM SP-2 [13].

P4T4-KT: four processes, one per machine with four application threads per process. Kernel threads are used throughout. Multiple application threads can be scheduled across processors in this case, and multiple requests can be handled by the *DSM_server* thread.

P4T4-UT: same as above, but using user level threads.
This was approximated by allowing only the default process level contention for the threads. These were then constrained to run on a single processor per node.

P4T4-KT-SIGIO: four processes, one per machine with four application threads per process with signal driven I/O. Instead of using polling, incoming messages generate a SIGIO signal, and a signal handler thread then processes the request.

P4T4-KT-SS: four processes, one per machine with four application threads per process. Compared to the P4T4-KT case, the *DSM_server* thread is changed to handle all requests sequentially, except for lock requests. As already mentioned, this was required to avoid a deadlock problem in the original Quarks runtime.

In every case, Release Consistency model was used.

3.2. SPLASH-2 programs

The data access patterns of the programs in the SPLASH-2 suite have been characterized in earlier research [11, 26]. FFT performs a transform of n complex data points and requires three all-to-all interprocessor communication phases for a matrix transpose. The data access is regular. LU-c and LU-n perform factorization of a dense matrix. The non-contiguous version has a single producer and multiple consumers. It suffers from considerable fragmentation and false sharing. The contiguous version uses an array of blocks to improve spatial locality. WATER-sp evaluates the forces and potentials occurring over time in a system of water molecules. A 3-D grid of cells is used so that a processor that owns a cell only needs to look at neighboring cells to find interacting molecules. Communication arises out of the movement of molecules from one cell to another at every time-step. Radix performs an integer radix sort and suffers from a high-degree of false sharing at page granularity during a permutation phase. WATER-n2 solves the same problem as WATER-sp, though with a less efficient algorithm that uses a simpler data-structure.

3.3. Image deblurring

The application tested is a parallel algorithm for deblurring of images obtained from Magnetic Resonance Imaging. Images generated by MRI may suffer a loss of clarity due to inhomogeneities in the magnetic field. One of the techniques for removing this blurring artifact is the demodulation of the data for each pixel of the image using the value of the magnetic field near that point in space. This method consists of acquiring a local field map, finding the best fit to a linear map and using it to deblur the image distortions due to local frequency variations. This is a very computation intensive operation and has previously been parallelized using a message passing approach [18]. Each thread deblurs the input image around its chosen frequency points and then updates the relevant portions to the final image. Since these portions can overlap, each thread does the update under the protection of a global lock.

3.4. Matrix multiplication

The matrix multiplication program uses a block-wise distribution of the resultant matrix. Each application thread computes a block of contiguous values, hence there is no false sharing.

4. Results

The overall execution times are shown in Figure 1, with results normalized to the total execution time of the *P16T1-KT* case. The data is separated into the time for:

Page Fault: the total time spent in the page-fault handler is adjusted for multiple overlapping faults on the same node.

Lock: the time spent to acquire a lock.

Barrier Wait: the time spent waiting on the barrier after completing Release Consistency related protocol actions.

Compute: this includes the compute time, as well as some miscellaneous components like the time spent sending diffs to other nodes, as well as the startup time.

The execution time for LU-c, LU-n, WATER-sp, and WATER-n2 can be seen to be dominated by the cost of synchronization events ie. barriers and locks. This is primarily due to the use of the release consistency model, which requires that all modifications to writable pages be flushed at synchronization points in the code. We are currently looking at improving the performance of this part of the system.

It can be seen that using multiple application threads on top of kernel threads can reduce the execution time in most case by up to 50 % as compared to using multiple processes. The primary reason is due to the reduction in the number of page faults, as can be seen in Figure 2. When multiple threads on a node access the same set of pages, they need to be faulted in only once.

Applications with a high computational component, like LU-c, RADIX, and MATMUL illustrate the problem with using only user-level threads. Each compute bound thread competes for the same processor of each machine, leading to very inefficient utilization. This can also be seen by the larger average barrier cost in case of MATMUL, since the first arriving thread has to wait for the remaining threads to finish computing.

The behavior of signal driven I/O compared to polled I/O can be explained by referring to Table 2. The overhead of signal generation becomes apparent as soon as the message size drops below 4 k bytes.

Table 3 shows that the concurrent server creates large number of threads for servicing DSM requests. The iterative server creates threads only for handling lock requests, the difference in counts is essentially due to paging activity and barrier calls. The advantage of using a concurrent *DSM_server* can be seen from the times taken for each protocol event, as shown in Figures 3, 4 and 5. The latency per event increases in general if an iterative server is used. This is particularly pronounced for the time per page fault. As expected, the lock time is almost the same, since both approaches use the same implementation. The discrepancy in RADIX is likely due to the fact that locks are used for flag based synchronization and there is a load imbalance. The barrier times are not affected markedly, which is expected, since at that time all the threads are idle anyway.

Additional runs were carried out with varying number of application threads to look at speed-up results.

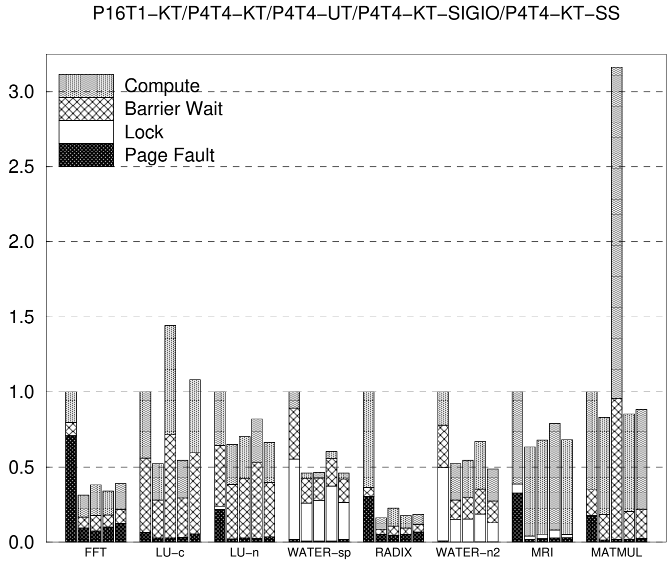

Figure 1. Execution Time Normalized to P16T1-KT

Program	Bytes	Messages	Avg. Size
FFT	36 M	7578	4.7k
LU-c	56 M	9516	5.8k
LU-n	10 M	2686	3.7k
WATER-sp	20 M	24045	831
RADIX	13 M	2825	4.6k
WATER-n2	13 M	95008	136
MRI	4 M	1319	3.0k
MATMUL	11 M	2553	4.3k

Table 2. Traffic per node

P1T1-KT: one process, on the UltraEnterprise 4000, ie. the sequential case.

P4T1-KT: four processes, one per machine, with a single application thread per process.

P4T2-KT: four processes, one per machine, with two application threads per process.

P4T4-KT: four processes, one per machine, with four application threads per process.

The results for LU-c, LU-n, MRI, and MATMUL are shown Figure 6. It can be seen that in most cases the execution time drops to half whenever the number of application threads is doubled. This also supports the notion of using multiple threads to increase the utilization of such SMP clusters.

5. Conclusions

Though the performance of each implementation can be seen to depend on the data sharing and communication pattern of the application program, some general trends can be observed. While going from the *P16T1-KT* base case to *P4T4-KT* the performance in general improves. This is primarily the result of using multiple kernel threads to reduce the cost of page-faults, lock acquisition and barriers. The *P4T4-KT-SIGIO* runs show that using a dedicated communication thread to poll for incoming messages is a preferred alternative to signal driven I/O. Though previous research has tried to use user-level thread

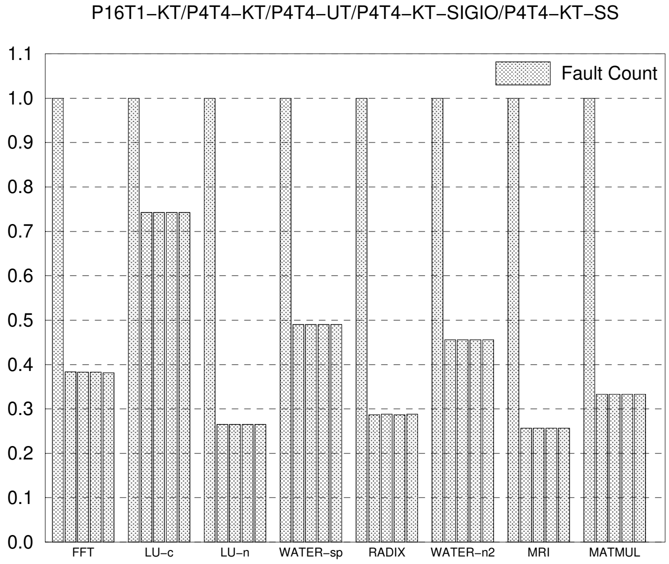

Figure 2. Page Fault Count Normalized to P16T1-KT

Program	Concurrent Server	Iterative Server
FFT	4626	9
LU-c	7027	9
LU-n	2079	9
WATER-sp	20050	90
RADIX	1905	39
WATER-n2	90784	30456
MRI	815	16
MATMUL	1373	12

Table 3. Threads Created per Node

packages to improve the performance of DSM systems, MATMUL and MRI clearly show that kernel threads provide far better performance. The concurrent *DSM_server* approach reduces the latencies for page-faults by allowing multiple requests to be handled concurrently. However the overall impact is not very high since the execution time is dominated by the barrier time in most applications.

When compared to message passing programs, additional sources of overhead for traditional software DSM systems have been identified to include separation of data and synchronization, overhead in detecting memory faults, and absence of aggregation [6]. Researchers have attempted to use compiler assisted analysis of the program to reduce these overheads. Prefetching of pages has been suggested by a number of groups for improving the performance of TreadMarks, by saving the overhead of a memory fault [19, 17]. This technique sacrifices the transparency of a page oriented DSM, but can be incorporated in parallelizing compilers. In *Strings*, a faulting thread does not block the execution of other application threads on the same process, hence the benefit of prefetching is not expected to be very large. Asynchronous data fetching was also identified to be a source of performance improvement [7]. In our system, the dedicated *DSM_server* and communication thread together hide the consistency related actions from the application threads.

Overall using kernel threads seems promising, especially for regular programs with little false sharing. Additional work needs to be done to identify the sources of overhead in the barrier implementation, since this dominates the execution time in most cases. Our current work is to improve the performance of the release consistency protocol. However for many applications, we have shown that using kernel-threads makes a significant difference in performance.

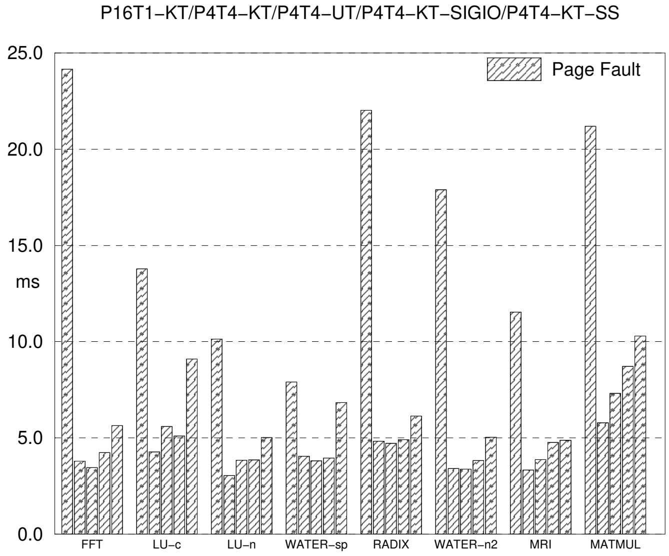

Figure 3. Page Fault Time

Acknowledgments

We would like to thank the anonymous reviewers, whose helpful comments shaped the final version of this paper. We also thank Padmanabhan Menon for providing the shared memory version of his MRI code.

References

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C.-W. Tseng. The SUIF Compiler for Scalable Parallel Machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, February 1995.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaeonopoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, pages 18–28, February 1996.

[3] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proceedings of the 1990 Conference on Principles and Practice of Parallel Programming*, pages 168–176, New York, 1990. ACM, ACM Press.

[4] B. N. Bershad and M. J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, Pittsburgh, PA 15213, September 1991.

[5] J. B. Carter. Design of the Munin Distributed Shared Memory System. *Journal of Parallel and Distributed Computing*, 1995.

[6] A. L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel. Evaluating the Performance of Software Distributed Shared Memory as a Target for Parallelizing Compilers. In *Proceedings of International Parallel Processing Symposium*, April 1997.

[7] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *ASPLOS-VII Proceedings*, volume 24, pages 186–197, Cambridge, Massachusetts, October 1996. ACM.

[8] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *ASPLOS-VII Proceedings*, volume 24, pages 210–220, Cambridge, Massachusetts, October 1996. ACM.

[9] E. W. Felten and D. McNamee. Improving the Performance of Message-Passing Applications by Multithreading. In *Proceedings of the Scalable High Performance Computing Conference*, pages 84–89, April 1992.

[10] B. Fleisch and G. Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages pp. 211–223, New York, 1989. ACM.

[11] D. Jiang, H. Shan, and J. P. Singh. Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, pages 217 – 229, Las Vegas, 1997. ACM.

[12] P. Keleher. *CVM: The Coherent Virtual Machine*. University of Maryland, CVM Version 2.0 edition, July 1997.

[13] P. Keleher and C.-W. Tseng. Enhancing Software DSM for Compiler-Parallelized Applications. In *Proceedings of International Parallel Processing Symposium*, August 1997.

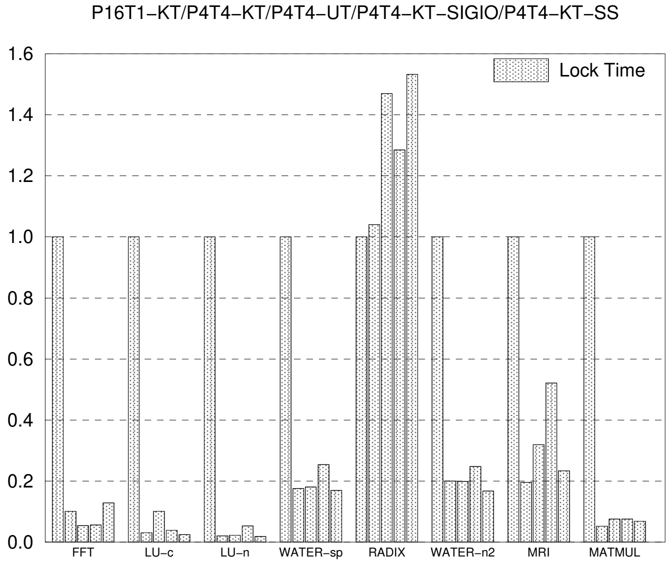

Figure 4. Time spent per Lock Acquisition, Normalized to P16T1-KT

[14] D. Khandekar. *Quarks: Portable Distributed Shared Memory on Unix*. Computer Systems Laboratory, University of Utah, beta edition, 1995.

[15] Kuck & Associates, Inc., Champaign, IL 61820. *KAP User's Guide*, 1988.

[16] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[17] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepol. Compiler and Software Distributed Shared Memory Support for Irregular Application. In *Proceedings of the Symposium on the Principles and Practice of Parallel Programming,*, 1997.

[18] P. Menon, V. Chaudhary, and J. G. Pipe. Parallel Algorithms for deblurring MR images. In *Proceedings of ISCA 13th International Conference on Computers and Their Applications*, March 1998.

[19] R. Mirchandaney, S. Hiranandani, and A. Sethi. Improving the Performance of DSM Systems via Compiler Involvement. In *Proceedings of Supercomputing 1994*, 1994.

[20] D. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin. Polaris: A New-Generation Parallelizing Compiler for MPPs. Technical Report 1306, Univ. of Illinois at Urbana-Champaign, CSRD, June 1993.

[21] S.-Y. Park, J. Lee, and S. Hariri. A Multithreaded Message-Passing System for High Performance Distributed Computing Applications. In *Proceedings of the IEEE 18th International Conference on Distributed Systems*, 1998.

[22] C. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten. Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 39–48, St. Charles, August 1989.

[23] E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the First USENIX Windows NT Workshop*, August 1997.

[24] K. Thitikamol and P. Keleher. Multi-threading and Remote Latency in Software DSMs. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, 1997.

[25] S. C. Woo, M. Ohara, E. Torri, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[26] Y. Zhou, L. Iftode, J. P. Singh, K. Li, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, pages 193 – 205, Las Vegas, June 1997.

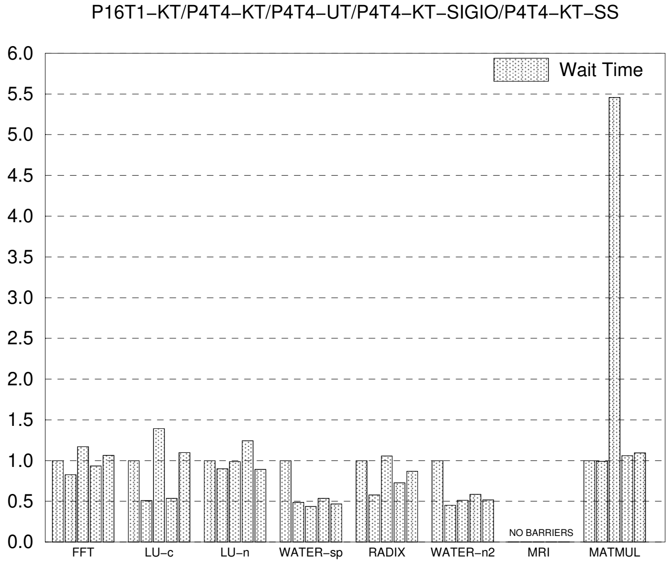

Figure 5. Time spent per Barrier Call, Normalized to P16T1-KT

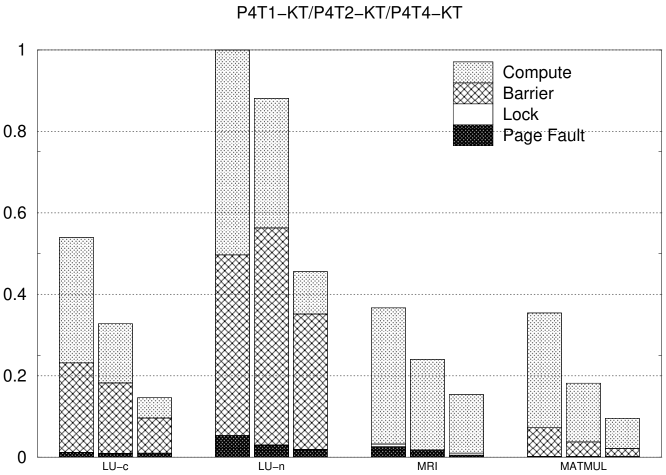

Figure 6. Execution Time Normalized to P1T1-KT