Thread Migration and Load Balancing in Non-Dedicated Environments

Kritchalach Thitikamol and Peter Keleher
University of Maryland
keleher@cs.umd.edu

Abstract

Networks of workstations are fast becoming the standard environment for parallel applications. However, the use of "found" resources as a platform for tightly-coupled runtime environments has at least three obstacles: contention for resources, differing processor speeds, and processor heterogeneity. All three obstacles result in load imbalance, leading to poor performance for scientific applications.

This paper describes the use of thread migration in transparently addressing this load imbalance in the context of the CVM software distributed shared memory system. We describe the implementation and performance of mechanisms and policies that accommodate both resource contention, and heterogeneity in clock speed and processor type. Our results show that these cycles can indeed be effectively exploited, and that the runtime cost of processor heterogeneity can be quite manageable. Along the way, however, we identify a number of problems that need to be addressed before such systems can enjoy widespread use.

1. Introduction

The realities of fast and cheap communication networks, combined with the emergence of the Internet as a commodity workspace, have led to a new emphasis on parallel and distributed applications and systems. Dedicated, homogenous parallel systems will always be an option, but the sheer number of cycles available in non-dedicated environments dwarfs those available in dedicated environments. The advent of fast commodity networks has made these cycles available.

The drawback is that these environments are usually heterogeneous in both resource capacities, such as otherwise-identical systems with differing clock speeds, and in terms of resource types, such as between Pentiums and Alphas. Moreover, such applications usually need to coexist with other applications running on the same systems. This is especially troublesome for tightly-coupled applications, i.e. those that communicate with fine granularity.

This paper presents a case study in the utility of such environments for running parallel programming systems normally associated with more tightly-coupled environments, such as SP-2's or clusters of workstations on the same high-speed LAN. We are specifically interested in the extent to which we need special operating system and programming model support. We evaluate mechanisms and policies that support automatic reconfiguration of software distributed shared memory (SDSM) applications in such heterogeneous environments.

We focus on SDSM applications in order to have a demanding application base. While some would argue that SDSM systems have little utility even in dedicated, homogenous environments, much less the heterogeneous environments that are investigating, we believe that long-term trends point the other direction. These trends are (i) the increasing ubiquity of small-scale shared-memory multiprocessors, (ii) the convergence between hardware and software implementations of DSM, and (iii) the similarity in application-restructuring principals needed for large-scale hardware DSM and small-scale software DSM.

First, dual and quad-processor shared memory machines are now appearing on desktops. As parallel, multithreaded applications become the norm rather than the exception, the ability to extend the same (or a similar) programming paradigm across network boundaries becomes more important. Second, the boundaries between hardware and software DSM are becoming more blurred. The FLASH [1] multiprocessor uses a protocol processor that looks and acts suspiciously like the user-programmable Lanai communications coprocessor used in Myrinet networks [2]. Finally, applications often need to be restructured in order to perform well on SDSM systems. However, the immediate goal of the restructuring is to improve data locality, precisely the same restructuring that needs to be done in order to get good performance on large-scale hardware DSMs [3], such as a 128-node SGI Origin.

We address two concerns with trying to exploit this type of environment. First, the parallel jobs may have to compete with other jobs for resources. Handling this type of contention is more complicated than merely scaling down the expected performance by the percentage of CPU cycles that the parallel job's process can be expected to get. Fine-grained parallel applications usually need each constituent process to be responsive, i.e. to handle incoming requests promptly. Such responsiveness is compromised if the parallel process is not scheduled when requests arrive.

Second, heterogeneity poses a whole slew of problems. We focus on systems with *heterogeneous capacities* in this paper, omitting discussion of *heterogeneous processor types* because of space considerations. Systems with heterogeneous capacities are binary-compatible machines with potentially differing clock rates, network interfaces,

	Pentium	Pentium Pro	Pentium II
Pentium	104 547	8107 1873 1287	2031 981 356
Pentium Pro	7793 1987 620	45 330	7654 1721 507
Pentium II	1650 1055 372	7702 1652 506	1357 647 227

Table 1: Remote request latency

and disks. This creates load-balancing problems for parallel applications that statically distribute work.

Despite these obstacles, there are reasons to be hopeful. We are attempting to exploit "found" resources. While high efficiency is desirable, any advantage that we obtain is worthwhile because the resources are otherwise idle.

Section 2 characterizes the applications, the environments, and the SDSM system that we will use. Section 3 discusses mechanisms and policies useful in implementing load-balancing through thread migration. Section 4 uses these mechanisms and policies to tolerate contention for resources, Section 5 does the same for environments consisting of machines with differing capabilities, and Finally, Section 6 concludes.

2. Applications, system, and environments

Our platform is the CVM software distributed shared memory system [4], modified to work in our target environment. The modified version of CVM supports multiple threads per node, thread migration, and heterogeneous sets of machines.

2.1 System characterization

All of our performance results are based on 100Mbit Fast Ethernet. Finding an acceptable network configuration was a non-trivial exercise. Although we have a number of high-performance networks in our department, all connect homogeneous sets of machines. While we have many fast machines, we have only a limited number of slower machines. We addressed both problems by using a small numbers (4) of processors connected by Fast Ethernet. While using fewer machines prevents us from studying scalability, we are still able to study the central issues of this paper: the ability of thread migration to address load-balancing problems created by both non-dedicated machines and unequal processor capacities. As such, the system configurations used in this paper are *conf-hom,* four 255 MHz Pentium II processors, and *conf-speed,* two 266 MHz Pentium II machines, one 200 MHz Pentium Pro machine, and one 133 MHz Pentium machine.

Table 1 shows the result of several benchmark tests. The boxes on the diagonal give costs of performing a `bcopy()` and `htonl()` on 8192-byte pages. Numbers off the diagonal show average round-trip latency seen by the requester for remote page, *diff*, and lock requests. A diff is a summary of the modified bytes on a single page. Lock requests are "1-hop", meaning that they are immediately satisfied by the destination rather than being forwarded. Note that links to and from the Alpha and Power2 machines are 10 Mbit/sec.

Apps	Description	Problem Sizes	Shared Pages
adi	ADI integration kernel	64K	2321
expl	Explicit hydrodynamics	512×512	2509
fft	3-D fast Fourier transform	64×64×128	3587
gauss	Gaussian elimination	2048x2048	2050
sor	Successive Over-Relaxtion	2048×2048	4097
tsp	Traveling salesman problem	19 cities	99
spatial	Spatial Water molecular dynamic	4096 mols	339
swm	Shallow-water model	512	2006
water	molecular dynamic simulation	512 mols	43

Table 2: Application characteristics

2.2 Application characterization

Our application suite consists of applications from a number of places. `water`, `spatial`, `barnes`, and `fft` are ubiquitous applications from the Splash-2 suite [5]. `tsp`, an implementation of the travelling salesman problem, and Gauss, which performs Guassian elimination with partial pivoting, are both from Rice. `expl` and `adi` are dense stencil kernels typical of iterative PDE solvers, parallelized by the SUIF [6] compiler. `swm` contains a mixture of stencils and reductions, and is from the SPEC benchmark suite. Table 2 summarizes input sets and shared segments sizes.

Figure 1 shows four-processor speedup of the applications on `conf-hom`, connected by UDP/IP over FastEthernet. The speedup of each is broken down into categories with size proportional to their contribution to execution time. The components are `comp`, the time spent running application code, `segv`, the time spent incurring

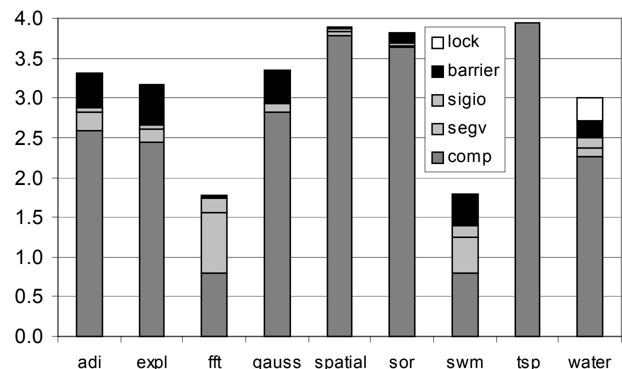

Figure 1: 4-Processor application speedup

and servicing segv signals (page faults), `sigio`, the time spent servicing remote requests, `barrier`, the average time spent waiting at barriers due to load imbalance, and `lock`, the time spent waiting for remote lock requests to succeed. The same categories will be used throughout this paper. Six of the eight applications get speedups of at least 3.0 on the four processors. The exceptions are `fft` and `swm`, which share large amounts of data and consequently have large `segv` times. The largest overhead category for most of the applications is `barrier`, which implies load imbalance. Note that this balance is not necessarily due to imbalance in the work assigned to nodes. Instead, imbalance is often created by unequal distributions of consistency actions, page faults either incurred or serviced. The largest overhead in `fft` and `swm` is caused is `segv`, corresponding to page faults. These applications share far more data than any of the others, and hence incur many more page faults.

We also ran the applications on UDP/IP over Myrinet [7]. This configuration made little difference, as the Myrinet IP stack is slower than FastEthernet for small messages.

Note that `tsp` is included primarily as an example of an application that balances load implicitly, and therefore does not need system-level load-balancing support.

3. Multi-threading and thread migration

Load balancing of iterative scientific codes is generally not necessary on dedicated, homogenous clusters. Scientific codes are usually balanced already, and dedicated clusters do not add any sources of imbalance.

However, all three environmental challenges considered in this paper, resource contention and heterogeneity of both capacity and processor type, result in differing execution rates across nodes. Since the majority of our applications use barrier synchronization in order to avoid data races, either the application or the system must balance load in order to use resources efficiently.

Reconfiguration of running applications is usually accomplished inside the program. For example, the `tsp` application discussed in this paper uses a centralized task queue that balances load implicitly. However, most scientific codes use a more static computation model.

We perform online reconfiguration transparently to the application via thread migration. Thread migration is the obvious choice because threads are visible to the system, and generally each has work statically assigned to it. Moving a thread, therefore, also moves the work. All of our applications can already be parameterized to run on different numbers of processors. Since sharing between threads is through a shared segment that is visible on all nodes, running with 8 threads on each of four processors can be made indistinguishable (except for performance) from having 1 thread on each of 32 processors.

To summarize: our runtime strategy is to: i) derive relative processor capacities through online measurement, ii) derive mappings of individual threads to nodes by using online data-sharing information, and iii) perform a single migration phase where threads are reshuffled.

Note that thread migration with homogenous processors is relatively easy, no type information is needed to translate object formats as data is passed between machines. Scientific applications like the ones discussed here usually have simple call graphs, and hence shallow stacks. The result is that thread migration is always cheaper than transferring a page.

3.1 Load balancing

Good load-balancing requires accurate information about either relative node capabilities, or of the relative amount of work each thread performs. Given one, the other can be derived from online measurement. With the exception of `tsp`, all of the applications in this study are iterative scientific workloads that partition work relatively equally among all threads. We can therefore use instrumentation of the first iteration of each application to drive our thread reconfiguration policy.

Given threads with equal amounts of work, we can derive relative processor capacities, c_i, as follows:

$$c_i = \frac{(bar - wait_i - os_i)}{T_i} \qquad (1)$$

where *bar* is the total time between a designated pair of barriers, $wait_i$ is the total amount of time spent waiting on remote requests or the second barrier to complete, and T_i is the number of threads on node *i*. The os_i variable is the amount of time spent by the operating system to implement communication calls, protection violation changes and faults, etc. Operating system trap costs can be measured directly, but the cost of calling signal handlers on protection violations, for example, need to be inferred from the number of occurrences and an average cost. On some systems, most notably AIX 3.2, the cost of protection changes and signal handlers can vary dramatically [8]. However, an average still gives a good guide.

Given relative node capacities, we can derive new thread distributions as follows. The number of threads that should be on node *i*, NT_i, can be expressed as:

$$NT_i = \frac{c_i}{\sum_{j=1}^{n} c_j} * \sum_{j=1}^{n} T_j \qquad (2)$$

where *n* is the number of nodes in the system.

One major problem with Equation (2) is that NT_i is not guaranteed to be an integer. However, for any set of new thread capacities, NT_i, integer solutions can be derived. Let *x* be the largest floating point divisor that goes evenly into all NT_i, and into differences NT_i - NT_j, for all *i*, *j*. The total number of threads needed with such a thread distribution is then:

$$tot = \sum_{i=1}^{n} \frac{c_i}{x} \tag{3}$$

Unfortunately, this number is likely to be large for any heterogeneous system, and large numbers of threads incur significant frictional costs. Figure 2 shows application performance as the number of threads per processor is increased from 1 to 8 threads per node on `conf-hom`. The slight increases in performance at small numbers of threads are due to latency hiding [9, 10]. Note that this increase would be much larger if we took startup costs into account, as in [10], but we consider only steady-state execution here. Additionally, we are not currently restructuring applications to avoid the problems discussed by Thitikamol [9], such as per-thread reductions, and duplication of thread state. These numbers impose a limit on the number of threads that we can use before performance degrades. Note that some of the lines are not smooth. The reason is that the pattern of sharing between threads can change with the number of threads, and communication (and therefore performance) generally correlates strongly with thread sharing. We therefore limit the total number of threads in our configurations to eight times the number of nodes, or 32 for the four-node configurations that we study. Integer NT_i values (the number of threads on node i) are derived by rounding. Note that the use of a small number of threads practically guarantees load imbalance.

While this approach requires relatively static sharing patterns and environments, much of the discussion in this paper is also applicable to more dynamic and irregular applications. Additionally, multiple rounds of measurement and reconfiguration could be used to accommodate long-running applications.

3.2 Thread mapping

Once the number of threads on each node has been determined, the actual threads to be mapped to that node need to be identified. In general, we would prefer to co-locate communicating threads. Thread-sharing patterns can be graphically illustrated using *correlation maps* [11]. Correlation maps are grids that summarize the number of pages shared between each pair of threads. Correlation maps can be shown graphically as two-dimensional squares, where the darkness of each point represents the degree of sharing between the two threads that correspond to the *x,y* coordinates of that point.

4. Contention for resources

Parallel applications running in non-dedicated environments must often compete for resources. This competition can affect overall parallel performance much more than would be indicated by a simple analysis.

Consider the situation where one of four processes of a parallel application runs on the same machine as an external sequential application. Assume each parallel process contains eight user-level threads. A simple analysis

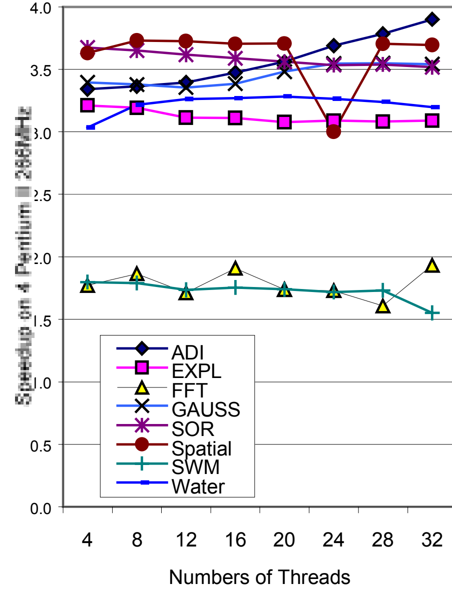

Figure 2: Speedup vs. total number of threads

might conclude that the parallel process that must compete with the sequential process would be slowed by a factor of two, in which case the thread distribution should be changed from 8-8-8-8 to 9-9-9-5, or 10-9-9-4. However, the situation might be far worse.

First, competing processes contend for resources other than the CPU, such as bandwidth.

Second, the parallel process's performance will suffer if it is not a strong competitor for the CPU. Consider a simple round-robin scheduler. A parallel job that communicates frequently would tend not to use its entire time slice, while a competing CPU-bound process might always do so. The CPU-bound process would obtain more cycles.

Finally, time-slicing might also hurt the parallel process's *responsiveness*. SDSM processes receive data and synchronization requests asynchronously. These requests need to be handled in a timely manner in order to avoid delaying the requester unduly. CVM's request handling is structured around signals being asynchronously generated when requests arrive. However, the signal is only delivered if the destination is currently scheduled. In the worst case, the message might be delivered only after all other active processes have used their entire time slices. This lack of responsiveness will directly degrade the performance of threads on other processors.

Figure 3 shows the slowdown from the dedicated (unloaded) case that occurs when a single processor is also hosting a sequential process that consumes 50% of the CPU when run in isolation. Figure 4 shows analogous data for a sequential process that consumes 100% of the CPU when run in isolation. For each application, we show a cluster of three bars showing slowdown i) if the thread

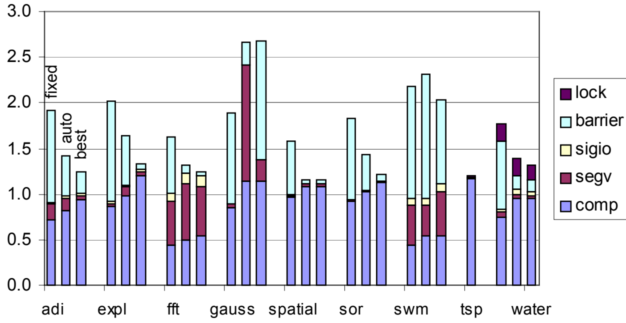

Figure 3: Homogeneous with one 50% loaded node

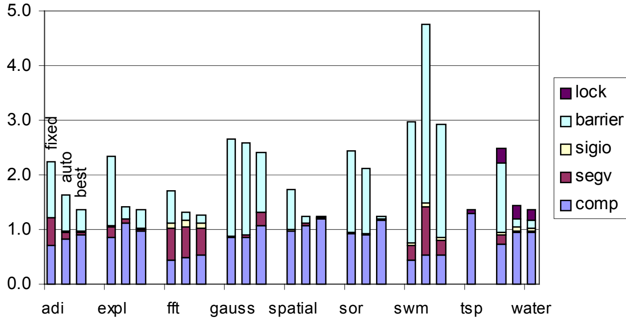

Figure 4: Homogeneous with one 100% loaded node

distribution is left unchanged (8-8-8-8, called "fixed"), ii) if we use a straightforward algorithm ("auto") that adjust the number of threads as in the example above and in Section 3, and iii) the slowdown with the "best" thread distribution. The best configuration is determined by running multiple candidate configurations and choosing the one with the lowest running time and is not intended to be representative of a strategy that can be used in a real system. Each bar is broken into the same five categories as Figure 1. Even with the 100% load, the automatic scheme produced a configuration almost as good as the best configuration for all but `sor` and `swm`.

We can calculate the best possible slowdown in the latter case assuming uniform work and communication. The total work needs to be divided into 3.5 pieces rather than 4, so the non-loaded processors each perform 28.6% of the work instead of 25%, an increase of 14%. Other than `gauss` and `swm`, actual slowdown ranges from 23% to 37%. The performance of the worst, `swm`, is because `swm` gets poor speedup to begin with, leading it to be a poor competitor versus the sequential load.

Table 3 shows the "auto" and "best" thread distributions whose performance is shown in Figure 3 and Figure 4. The best distribution generally places fewer threads on the loaded node than the "auto" distribution, presumably because of the reasons discussed at the beginning of this section. Nonetheless, the auto approach performs significantly better relative to the best possible distribution than with non-dedicated environments.

5. Heterogeneous processor capacities

Clusters of machines with heterogeneous processor capacities naturally arise in environments where machines are bought frequently. Machines purchased just weeks apart often have significantly different clock rates. Requiring that clusters be completely homogenous, therefore, would eliminate many opportunities for parallelism.

Our heterogeneous capacity configuration, `conf-speed`, consists of two 266 MHz PentiumII's, a 200 MHz Pentium Pro, and a 133 MHz Pentium. On average, the latter two machines have a capacity of 85% and 38% of the PentiumII's, leading to an average potential speedup of 3.23 over sequential execution on one of the PentiumII's. This figure assumes that the application scales linearly to begin with, so realizable speedups are smaller.

Figure 5 shows speedups relative to PentiumII executions for two thread distributions: the auto distribution implied by processor capacity and the "best" distribution, as in Section 4. Most of the applications perform well, with the exception of `fft` and `swm`. These two applications again have large bandwidth requirements. Matters are far worse than for `conf-hom` because execution time is dominated by page faults, which are very slow when directed at one of the slower machines.

6. Implications and conclusions

Many researchers have proposed ways to exploit otherwise-idle resources in non-dedicated environments [12]. However, most investigated sequential or coarse-grained distributed applications because of the inherent overheads in such environments.

Conventional wisdom holds that tightly coupled ap-

		adi	expl	fft	gauss	spatial	sor	swm	water
50% non-dedicated	best	10-10-10-2	11-10-10-1	10-10-10-2	10-10-9-3	9-9-9-5	10-10-10-2	10-10-9-3	10-10-10-2
	auto	9-9-10-4	9-9-10-4	9-9-9-5	10-9-9-4	9-9-9-5	9-8-9-6	10-10-10-2	10-9-9-4
100% non-dedicated	best	10-10-9-3	9-9-9-5	10-10-10-2	10-10-10-2	10-9-9-4	10-10-9-3	10-10-9-3	10-10-10-2
	auto	9-9-10-4	10-9-10-3	9-9-9-5	8-8-8-8	9-9-9-5	8-8-9-7	10-10-9-3	10-9-9-4
heterogeneous processor capacity	best	9-9-5-9	11-10-2-9	11-10-4-7	10-9-4-9	11-10-4-7	11-11-1-9	10-10-3-9	10-11-4-7
	auto	9-9-5-9	10-9-3-10	11-11-3-7	9-9-4-10	11-11-4-6	11-11-1-9	10-10-3-9	11-10-4-7

Table 3: Thread distributions

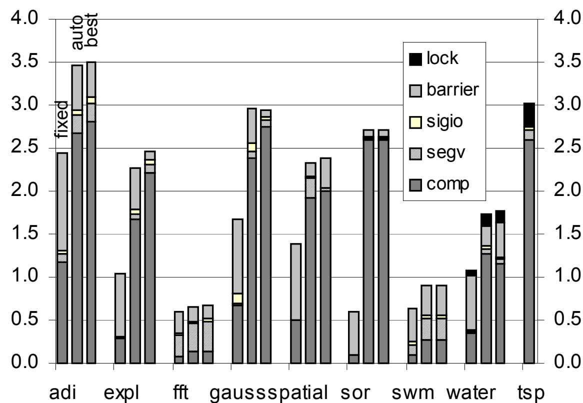

Figure 5: Speedup with heterogeneous speeds (relative to fastest node)

plications can only be profitably parallelized on sets of roughly comparable machines that are connected by fast networks. However, in one case we were able to obtain performance improvement even from a machine five times slower than the fastest in the configuration. The key to evaluating performance in this environment, however, is in remembering that these resources *are* otherwise idle. Any use of found resources that improves execution time is well spent; linear parallel speedup is not necessary.

This paper has described the performance of nine demanding applications running on top of a modified version of the CVM SDSM in such environments. We separate the key performance challenges into three categories: contention for the CPU resource, heterogeneous processor capacity, and heterogeneous processor types. Most of the applications achieved good speedup relative to the fastest constituent node, despite the tight coupling of processes and fine-grained communication required by the SDSM system.

CVM performs well by balancing load through thread migration. The automatic thread distribution mechanism performed well for all but a few of the applications running in non-dedicated environments. Our future work will include investigation into the use of multiple rounds of reconfiguration as a means of narrowing the gap even further.

A second major issue that needs to be addressed is the use of SDSM systems in heterogeneous environments. Though this issue has been addressed before, no study has is both current and comprehensive. We believe that by adding several restrictions to the programming model we can minimize the direct costs of heterogeneity. It remains to be seen whether these restrictions can be eased without hurting performance, but the current programming model would require very few changes to our suite of nine applications.

7. References

[1] J. Kuskin and D. O. e. al., "The Stanford FLASH Multiprocessor," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

[2] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su, "Myrinet: A Gigabit-per-second Local Area Network," *IEEE Micro*, vol. 15, pp. 29-36, 1995.

[3] D. Jiang and J. P. Singh, "Scaling Application Performance on a Cache-coherent Multiprocessors," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[4] P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models," in *Proceedings of the 16th International Conference on Distributed Computing Systems*, 1996.

[5] S. C. Woo, M. Ohara, E. Torrie, 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*, June 1995.

[6] R. P. Wilson, R. S. French, C. S. Wilson, J. M. Amarasinghe, S. W. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An Infrastructure for research on parallelizing and optimizing compilers," *ACM SIGPLAN Notices*, vol. 29, pp. 31-37, December 1994.

[7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovi, and W.-K. Su, "MYRINET: A Gigabit Per Second Local Area Network," *IEEE-Micro*, vol. 15, pp. 29-36, 1995.

[8] P. J. Keleher, "Update Protocols and Iterative Scientific Applications," in *The 12th International Parallel Processing Symposium*, March 1998.

[9] K. Thitikamol and P. Keleher, "Per-Node Multithreading and Remote Latency," *IEEE Transactions on Computers*, vol. 47, pp. 414-426, April 1998.

[10] T. C. Mowry, C. Q. C. Chan, and A. K. W. Lo, "Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory," in *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[11] K. Thitikamol and P. J. Keleher, "Active Correlation Tracking," in *The 19th International Conference on Distributed Computing Systems*, June 1999.

[12] M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," in *International Conference on Distributed Computing Systems*, 1988.