Active Correlation Tracking

Kritchal Thitikamol Peter J. Keleher
(kritchal|keleher)@cs.umd.edu

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

We describe methods of identifying and exploiting sharing patterns in multi-threaded DSM applications. Active correlation tracking *is used to determine the affinity, or amount of sharing, in pairs of threads. Thread affinities are combined to create correlation maps, which summarize sharing between all pairs of threads in the application.*

Correlation maps can be used in two ways. First, they can be used as an aid for performance tuning, Second, they can be used to estimate the impact on communication requirements of reconfiguring running applications through thread migration. Thread migration provides a way of tuning applications for which sharing information is not known a priori, and a means of adapting to dynamic algorithms or environments.

We show that i) accurate thread affinities can be obtained without multiple rounds of migration, ii) thread affinities lead to good approximations of application communication requirements, iii) simple heuristics can use thread affinities to efficiently approximate optimal mappings of threads to nodes, and iv) good placement is essential for high performance.

1. Introduction

This paper describes *active correlation tracking*, a mechanism for tracking data sharing between threads, and its implementation in CVM [1], a software distributed shared memory (DSM) system. DSMs are software systems that provide the abstraction of shared memory to threads of a parallel application running on networks of workstations. Consistency is maintained by using virtual memory techniques to trap accesses to shared data and ensure consistency. Information on the type and degree of data sharing is useful to such systems because the majority of network communication is caused by the underlying consistency system. When a pair of threads located on distinct machines (nodes) both access data on the same shared page, network communication can only be avoided by moving at least one of the threads so that they are located on the same node.

In order to minimize communication, therefore, the system needs to identify the thread pairs that will cause the most communication if not located on the same node. The information should be complete, in that we need information on all threads in the system, and it must be accurate, in that small errors in the relative ordering of thread pairs might cause large differences in communication.

Ideally, sharing behavior would be measured in terms of access rates. More specifically, we can define a density function that represents the access rate of thread *i* to page *p*. The correlation of two threads over page *p* can be defined to be the product of the density function of the two threads for page *p*. The overall correlation of two threads is then just the sum of the correlations of the threads over all shared pages in the system [3]. However, the notion of an access *rate* is difficult to capture in a DSM. Once a page has been mapped locally, subsequent accesses to the page proceed transparently. Hence, we can not track the rate of individual accesses. A rough estimate might be obtained by tracking the average length of time a given page remains invalidated before being revalidated. Unfortunately, this estimate can be greatly affected by intervening events. For instance, 100 *u*secs is a long interval if it contains only local accesses. However, a remote access can take milliseconds. Such events make it unlikely that the rate of page revalidation would accurately reflect the access rate. Systems that capture shared writes through binary rewriting [4] rather than page faults could presumably capture accurate densities. One major drawback of this approach is that a naïve implementation would add overhead to all writes, not just those that occur when the tracking mechanism is turned on. Function cloning could be used to create tracking and non-tracking versions, but every function that might possibly access shared data would have to be cloned.

Current systems [2, 3], therefore, merely track the set of pages that each thread accesses. Changes in sharing patterns are usually accommodated through the use of an aging mechanism.

In any case, word-level access densities are not the proper abstraction for a page-based system. We therefore track data sharing between threads by correlating the threads' accesses to shared memory. Two threads that frequently access the same shared pages can be presumed to share data. We define *thread correlation* as the number of pages shared in common between a pair of threads. We define the *cut cost* to be the aggregate total of thread correlations for thread pairs that must communicate across node boundaries. Cut costs can then be used to compare candidate mappings of threads to nodes in the system. Once the best mapping has been identified, the run-time system can migrate all threads to their new homes in one round of communication.

Application Names	Types of Synchronization	Input sizes	Shared Pages
Barnes	barrier, lock	8192 bodies	251
FFT6	barrier	64×64×64	1796
FFT7	barrier	64×64×128	3588
FFT8	barrier	64×64×256	7172
LU1k	barrier	1024×1024	1032
LU2k	barrier	2048×2048	4105
Ocean	barrier, lock	256 oceans	3191
Spatial	barrier, lock	4096 mols	569
SOR	barrier	2048×2048	4099
Water	barrer, lock	512 mols	44

Table 1: Application Characteristics

In the sections that follow, we show that thread correlations can be used to predict communication cost (Section 2), and to visualize and model sharing behavior (Section 3). We then describe an efficient mechanism for deriving thread correlations (Section 4) and provide examples of its use (Section 5).

Our experimental environment consists of the CVM [1] software DSM system, running on a cluster of eight workstations. Each workstation is equipped with 194 MBytes of RAM and a 266 MHz Pentium II processor. The workstations run the Linux operating system, version 2.0.32, and are connected by a Myrinet [5] commodity network. The shared memory applications that we use are summarized in Table 1. They include Barnes, FFT, LU, Ocean, Spatial, and Water from the SPLASH-2 benchmark suite [6], as well as a simple successive-over-relaxation (SOR) application. Unless otherwise specified, all runs include sixty-four threads divided equally among eight nodes.

This paper implicitly assumes that multiple threads are running on each node. This is desirable for several reasons. First, it allows the programming model to be decoupled from the hardware model. The application's structure can be independent of any specific hardware environment. Second, multiple local threads allow the system to use context switching to hide remote latencies [7, 8]. Finally, dividing the work on any single node into multiple units allows the system more freedom in redistributing or balancing the work through thread migration.

2. Thread correlations and cut costs

The cut cost of a given mapping of threads to nodes is the pairwise sum of all thread correlations, i.e. a sum with n^2 terms, where n is the number of threads. This sum represents a count of the pages shared by threads on distinct machines.

We hypothesize that cut costs are good indicators of data traffic for running applications. We tested this hypothesis experimentally by measuring the correlation between cut costs and remote misses of a series of randomly generated thread configurations. A remote miss occurs any time a process accesses an invalid shared page. Pages are invalid either because the page has never been accessed locally, or because another process is modifying the page[1]. In either case, the fault is handled by retrieving a current copy of the page from another node. For purposes of this experiment, we assume that all remote sites are equally expensive to access; thereby ensuring that the number of remote faults accurately represents the cost of data traffic.

Apps	Slope	Y-intercept	Correlation Coefficient
Barnes	0.227	-14483.4	0.742
FFT7	2.517	-23506.9	0.925
FFT8	2.805	-16275.6	0.911
LU2k	2.694	-76837.3	0.724
Ocean	4.508	-92112.1	0.937
Spatial	0.079	-2760.1	0.458
SOR	4.100	-21.4	0.961
Water	0.402	-3011.4	0.779

Table 2: Remote misses as a function of cut costs

The first step of this experiment was to collect thread correlations. We computed thread correlations by running 64-thread versions of each of our applications, tracking the pages accessed by each thread (more detail on the procedure that we used is presented in Section 4.2), and enumerating the set of pages shared by each thread pair.

Given a complete set of thread correlations, we generated 300 random thread configurations (placements of threads on nodes) for each application. Equal numbers of threads were not necessarily present on each node, although no node ever ended up with fewer than two threads. Unequal numbers of threads might be desirable in the presence of heterogeneous node capacity, whether due to competing applications or simply because some machines are faster than others. We ran the applications with each configuration, recording the number of remote misses that occurred. The results are summarized in Table 2 and shown graphically in Figure 1. In all cases except Spatial, correlation coefficients are at least 0.72. Aside from a single outlier caused by the garbage collection mechanism, SOR's correlation coefficient would be 1.0.

There are at least three good reasons why a strictly linear relationship between cut costs and remote misses might not hold. First, as noted above, correlation tracking does not capture fault rates. Differing synchronization patterns might cause applications to suffer multiple faults on some shared pages, but only single faults on others. For example, assume that threads t_1 and t_2 are on distinct nodes, and that t_1 modifies page x after barriers 1 and 3, and page y after barrier 3. Further, assume that thread t_2 reads from page x after barriers 2 and 4, and reads from y after barrier 4 as well. Thread t_2 will have two remote faults on page x, but only one on page y. However, both pages will count equally in the calculation of thread correlations.

Second, the order that local threads read and write shared pages is often non-deterministic. This non-determinacy does not affect correctness of an otherwise correct program, but it could easily lead to a situation in which a page is invalidated between two local accesses. The result would be an extra remote fault.

[1] This is a gross simplification, but captures the essence.

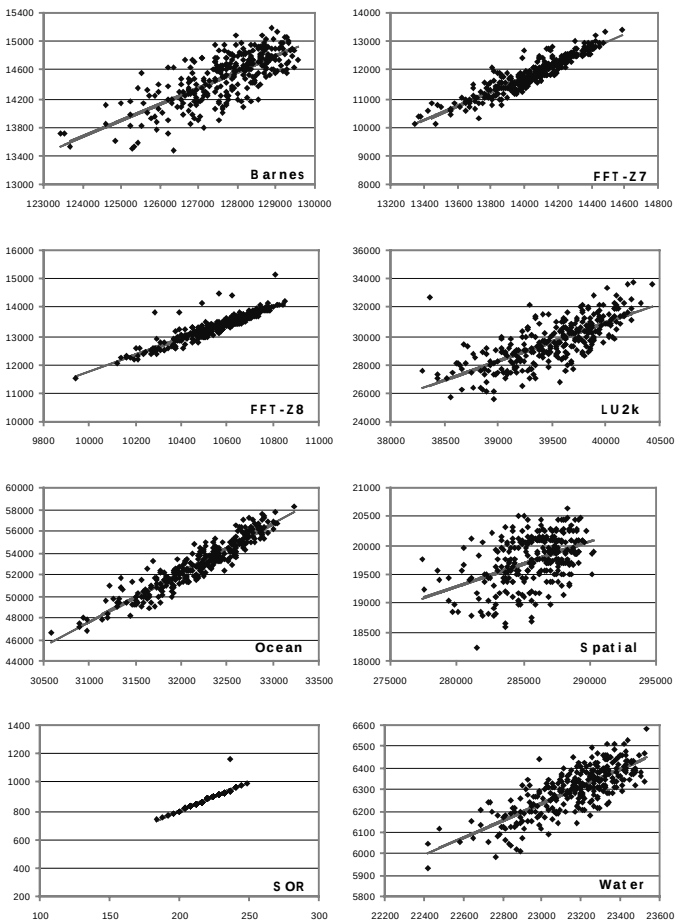

Figure 1: Cut costs versus remote misses: Cut costs are on the *x* axis, remote misses are on the *y*.

Finally, the DSM protocol used in our experiments requires periodic garbage collections. Garbage collections consolidate all modifications of a single page at a single site, often requiring multiple remote fetches to do so. Other replicas of these "collected" pages are invalidated, rather than being updated. These invalidations lead to additional remote faults. We have verified that garbage collection is not the primary culprit in deviations from an ideal linear relationship in any of the above applications, but it is a contributing factor.

3. Correlation maps

This section describes the use of thread correlations in creating *correlation maps.* Correlation maps are grids that summarize correlations between all pairs of threads. We can represent maps 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. Table 3 shows correlation maps for each application with 32-thread, 48-thread, and 64-thread configurations.

Correlation maps are useful for visualizing sharing behavior. For example, note the prevalence of dark areas near the diagonals in most of the applications in Table 3. These areas represent nearest-neighbor communication patterns. SOR has no other sharing traffic at all. Water, on the other hand, has nearest-neighbor traffic that starts high, smoothly decreases, and then increases with "distance" between the threads in each pair.

The sharing in LU and FFT, on the other hand, is concentrated in discrete blocks of threads, rather than being continuous. For example, note the 8 by 8 sharing structure of the 32-thread configuration of the LU2k application in Table 3. The correlation map shows that the majority of the sharing occurs in the 8 by 8 blocks. One implication of this sharing pattern is that an eight-node configuration would probably have much more communication than a four-node configuration. A balanced, eight-node configuration would place 4 of the 32 threads on each node. However, any such configuration would entail breaking up the large sharing blocks, implying that an eight-node configuration would have much more communication than a four-node configuration. We have confirmed that this is the case. In fact, the communication difference turns out to be enough to make the eight-node configuration slower than the four-node configuration on some clusters of machines in our testbed. Correlation maps by themselves, however, do not provide enough information to have determined this without running both configurations.

3.1.1 Variation with number of threads

The columns in Table 3 correspond to 32, 48, and 64 threads, respectively. Somewhat surprisingly, they show that sharing characteristics can significantly vary with the number of threads.

The overall structure of the correlation maps for SOR, Barnes, and Water are little affected by changing the number of threads. This could have been predicted from a cursory examination of application structure. Threads in SOR, for example, share only single rows of data between pairs of adjacent threads. Hence, changing the number of threads does not affect the structure of the map. Water and Barnes are more complex, but still easily explained.

However, FFT and Spatial both show distinct irregularities at 48 threads. This reflects applications that expect the number of threads to be a power of two, and are unable to properly balance load across other system configuration.

Although FFT's configurations for 32 and 64 threads superficially appear similar, they reflect sharing blocks of four and eight threads, respectively. Nonetheless, the implication is that communication behavior remains static across four- and eight-node configurations.

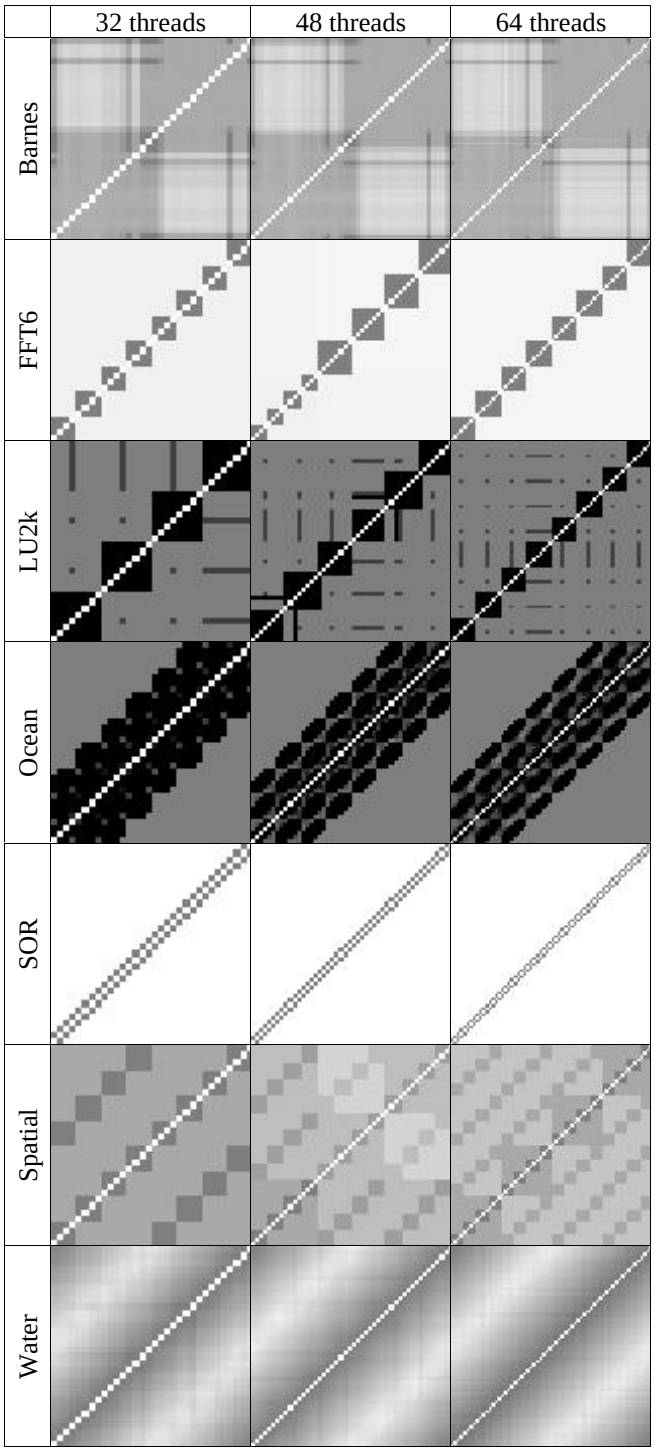

Table 3: Correlation Maps – Increasing thread correlation is represented by darker shades. Each map is *nxn*, where *n* is the number of threads. The origin is in the lower left.

Ocean and Spatial both change significantly from 32 to 64 threads. With Ocean, discrete blocks of threads have nearest-neighbor communication. Increasing the number of threads increases the size of these blocks, but not their count. Spatial's behavior is the result of phases with distinct sharing patterns. One of the phases moved from 8 blocks of 4 threads to 4 blocks of 16, while the other moved from 8 blocks of 4 to 16 blocks of 4.

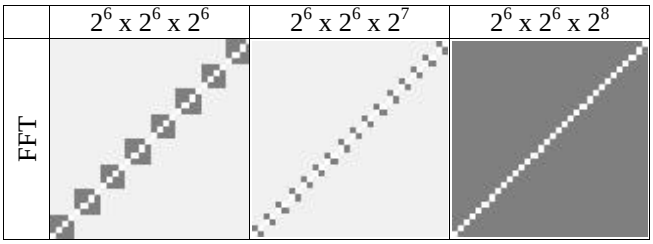

Table 4: 64-thread FFT versus input set

3.1.2 Variation with input

Several of the applications' correlation maps also varied with input set. Correlation maps for 64-thread versions of FFT with three different input sets are shown in Table 4. With an input of 2^6 x 2^6 x 2^6, sharing is organized into eight eight-thread clusters, with heavy sharing within clusters but little outside. Doubling the input set size to 2^6 x 2^6 x 2^7 broke the sharing into 32 disjoint four-thread blocks, with background inter-block sharing significantly reduced from the smaller input set. Finally, doubling the input set size again to 2^6 x 2^6 x 2^8 resulted in uniform all-to-all sharing.

Changes in the sharing of other applications are less striking. The majority varied only in relative intensity rather than in structure. Note that even this type of change might affect the choice of optimal thread mapping.

4. Correlation-tracking mechanisms

4.1 Passive correlation tracking

Previous systems obtained page-level access information by tracking existing remote faults. Remote faults occur when local threads attempt to access invalid shared pages. Remote faults are satisfied by fetching the latest version of the shared page from the last node that modified it. The underlying DSM can overload this process to inexpensively track the causes of remote faults, slowly building up a pattern of the pages accessed by each thread.

The problem is that there is often more than a single thread running on each machine, and these threads share state. Once the first thread on a node validates a given page through a remote fault, all other local threads can access the page without invoking the DSM system.

Hence, the system only gains partial information about the sharing behavior of local threads. Any migration decisions are made with only partial information, often leading to bad long-term choices. These bad choices are discovered only after the threads have been migrated to other nodes. Once a thread migrates off of a local host, the interactions between that thread and those left behind become visible in the form of remote faults (unless masked by the actions of other threads on the new node). These faults may identify threads that should be moved back to their original positions, resulting in ping-ponging of threads across the system.

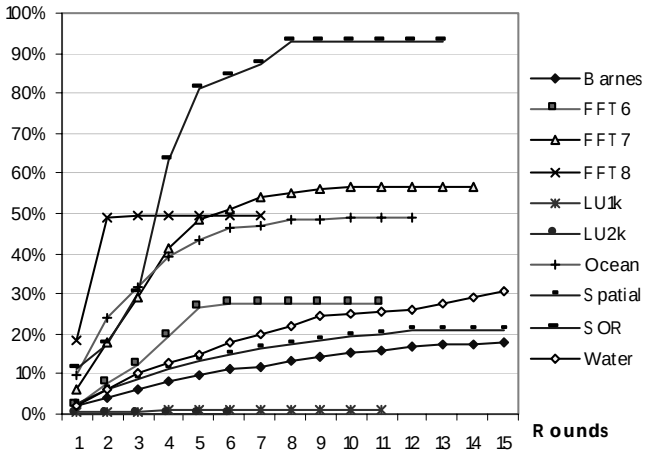

Figure 2: Passive Information-Gathering

Figure 2 shows the percentage of complete sharing information gathered by the passive tracking approach as a function of the number of migration rounds. Even at the end of the migrations, the passive tracking only comes close to obtaining complete information for SOR, by far the least complex of our applications. Each round consists of gathering page fault information for an iteration of the application, followed by migrating threads to new locations.

The applications averaged slightly more than six rounds of migrations before stabilizing, although Figure 2 shows all rounds in which new information is gained. The term "stabilizing" is used advisedly. Recall that passive correlation tracking only learns about the first local thread to access a page during any synchronization interval. This means that the speed at which information is accumulated is non-deterministic. A configuration might appear optimal for several iterations before the non-deterministic scheduling of threads reveals new information. This happened for Water, where migrations occurred eight times, followed by three iterations in which no better configurations were found, followed by five more iterations in which new information caused additional rounds of migrations to occur.

4.2 Active correlation tracking

Network ping-pongs can be avoided by obtaining additional information about correlations between local threads before any thread migration takes place. We obtain this information through an *active correlation-tracking* phase, which iteratively obtains access information for each local thread. The correlation-tracking phase spans a single iteration of each of the applications. The algorithm uses two data structures on each node: per-page correlation bits, and per-thread access bitmaps:

1. At the start of the tracking phase, all pages are read-protected and the *correlation bit* of each page is set. The pages' previous states are saved in CVM's page structures. The thread scheduler is placed in a special mode that prevents thread-switching from occurring until the next barrier has been reached.
2. At each access fault for a page whose correlation bit is set (a *correlation fault*), the corresponding bit in the per-thread *access bitmap* is set, and the correlation bit is reset. The page is then returned to its original state and the fault handler returns. If the access type would have caused a violation even outside the correlation-tracking phase, an additional fault occurs and is handled normally.
3. At the next barrier, the system switches to the next thread, sets all correlation bits again, and once again read-protects all pages. This thread is then allowed to proceed in the same manner as the previous thread.
4. The tracking phase ends when all threads reach the next barrier. At the end of the correlation-tracking phase, all correlation bits are reset and untouched pages are returned to their previous protection state.

After the tracking phase has ended, the per-thread access bitmaps specify exactly which pages each thread accessed during the tracking phase.

The tracking phase has two primary forms of overhead. The most obvious is the cost of the correlation faults. This cost scales with the number of pages accessed locally, and the degree of sharing between the local threads. Given a system with n nodes and p pages, the local threads will usually access at least p/n pages, more if there is a large amount of data sharing between threads. Local sharing increases the number of faults because each shared page incurs more than one page fault. However, the cost of correlation faults on distinct nodes is incurred in parallel.

The second cost results from disabling the thread scheduler during the tracking phase. The scheduler is disabled so that each thread can be run from one barrier to the next atomically with respect to the other local threads. This decreases the overhead of the mechanism because each thread switch that occurs while tracking is enabled requires all page protections to be restored to the protections specific to the new thread. However, turning off the thread scheduler eliminates the latency toleration advantages of per-node multi-threading. The performance impact of losing this amount of latency toleration is usually on the order of 10-15% [7], and is only incurred during the active correlation-tracking phase.

Table 5 shows the cost of performing the active correlation tracking with eight threads per node. The first three columns show iteration times without and with correlation tracking, and the percent slowdown from one to the other. Two of the applications, Ocean and SOR, produced slowdowns of more than 50%. 'LU2K' slowed down by one third, and the others by less than 12%. This slowdown refers only to the slowdown for the tracked iteration.

While not prohibitively expensive even on these applications, the tracking process is too costly to perform often. However, the iterative nature of our applications allows us

to perform the correlation tracking only once, amortizing the cost over the rest of the computation. For example, the above overheads might be tolerable if amortized across ten iterations, and would certainly be tolerable if each application performed 100 iterations. In fact, amortized slowdown was less than 1% for all of our applications except Ocean.

As noted above, all overhead of the tracking phase is incurred locally, and in parallel across nodes of the system. This implies that the absolute runtime cost of the tracking phase should not increase as the number of nodes is increased. This is in contrast to the passive ping-ponging approach, in which increasing system size would probably increase the number of thread migrations.

The absolute cost of this tracking phase *is* sensitive to the overall amount of sharing in the system. Since sharing means that multiple threads are accessing the same pages, such sharing increases the total number of segmentation violations. Systems with little or no sharing are therefore insensitive to the number of threads. However, as sharing increases, the number of threads can become significant.

The last three columns of Table 5 show the total number of tracking and coherence faults incurred during tracked iterations, and a measure of how efficient the correlation tracking is at obtaining access information. "Sharing degree" is a count of the total number of distinct shared pages that were accessed during the tracked iteration, divided by the number of induced tracking faults. The result gives the average number of local threads that access distinct shared pages that are touched locally. For example, if t_1 accesses page x, t_2 accesses x and y, and t_3 accesses y and z, then the average number of threads accessing each locally accessed page is 1.7 (average of 1, 2, and 2). SOR's sharing degree is 1.08, accurately reflecting the fact that only boundary rows are shared. Water, on the other hand, has an average sharing degree of 6.8, meaning that for each node, an average of 6.8 out of the 8 local threads touched every shared page that was accessed by any of them. We explain this high degree of sharing by noting that Water is a molecular simulation in which force computations for any given molecule usually reflect the influence of most of the rest of the molecules.

Appls	Iteration time (secs)		Percent Slowdown	Faults		Sharing Degree
	Off	On		Tracking	Coherence	
Barnes	2.24	2.32	3.62%	8628	8316	6.583
FFT6	0.37	0.40	8.99%	5216	928	2.657
FFT7	0.67	0.75	11.28%	6112	1824	1.734
FFT8	1.41	1.51	7.32%	5600	5920	1.268
LU1K	0.30	0.32	8.11%	9855	232	7.359
LU2K	0.80	1.06	33.33%	36102	344	7.821
Ocean	1.92	3.26	69.92%	62039	12439	2.112
Spatial	13.43	13.60	1.27%	38286	6296	6.030
SOR	0.15	0.26	75.68%	8640	56	1.081
Water	1.07	1.09	2.25%	2983	1427	6.754

Table 5: 64-Thread Tracking Overhead

5. Using correlation maps to direct migration

Sections 2 and 3 showed that thread correlations can be used to model communication behavior, and Section 4 showed how to efficiently derive this information online. This section discusses uses of this information.

Thread correlations are primarily useful as a means of evaluating cut costs (and, indirectly, communication requirements) of candidate mappings of threads to nodes. Such comparisons are only meaningful if applications can be configured to match arbitrary thread mappings. Given that we derive our correlation mappings online, such configurations must be of running applications.

Hence, reconfigurations require thread migrations. We assume a DSM system that supports per-node multithreading [7] (multiple threads per node) and thread migration. Per-node multithreading is only problematic when DSMs only allow dynamically allocated data to be shared, like CVM. The problem is that it exposes an asymmetry in the threads' view of data. Threads on a single node share the same copy of statically allocated global data, but each node has distinct copies. This problem is usually handled by restricting threads from accessing any of these variables. Instead, threads can access only stack and globally shared data.

Given the above, thread migration can be accomplished through little more than copying thread stacks from one machine to another. Care must be taken to preserve the stack's address before and after a copy so that pointer values do not become orphaned. Additionally, thread migration in systems that support relaxed consistency models must ensure that the thread's view of shared data at the destination is not missing any updates that were visible at the source.

Systems that have a large number of threads per node might allow nodes to unilaterally export threads to other nodes. Load balance can only be maintained, however, if the number of exported threads matches the number imported[2]. Good decisions about which thread(s) should be imported usually require global information, and do not change the number of threads on any node.

Online reconfiguration may be necessary or desirable for a number of reasons. As an example, consider Figure 3 (a). This correlation map represents a version of FFT with 32 threads distributed equally across four nodes. The points inside the dark squares represent those thread pairs that are located on the same nodes, and hence do not figure into cut costs or require network communication. There are four squares, since there are four nodes, or regions where sharing is free. Since all of the dark regions are inside the "free zones" that represent nodes, we can infer that communication requirements will be relatively minimal.

Now consider instead Figure 3 (b). This picture represents a configuration of four threads running on each of eight nodes. The correlation map is the same, but the smaller "free zones" encompass only half of the dark areas. Hence, we can infer that this configuration has more communication than the four-node version. Together with information on the ratio of communication to computation in the application, a runtime system could potentially make a

[2] Assuming that threads have equal work.

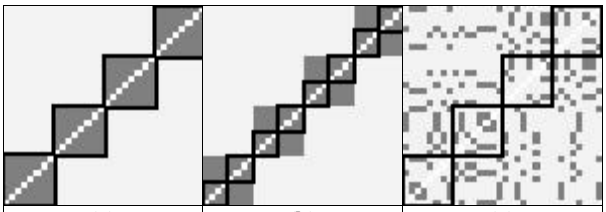

Figure 3: 32-thread FFT, 2^6 x 2^6 x 2^6 - (a) on four nodes, squares indicate thread sharing that does not cause network communication, (b) on eight nodes, as above, (c) randomized thread assignments for four nodes

rough guess at whether the eight-node configuration would have any performance advantage over the four-node version.

Finally, consider Figure 3 (c). This is the same application, with unchanged sharing patterns. However, we have randomly permuted the assignment of threads to nodes. Doing so results in a configuration with a much higher cut cost, which is not addressed effectively by either the four-node or eight-node configurations. Similar situations would arise with applications in which sharing patterns change slowly over time.

5.1 Identifying good thread assignments

The combination of finding the optimal mapping of threads to nodes is a form of the *multi-way cut* problem, and is NP-hard. While good approximation schemes have been found for the general form of the communication minimization problem [9], our problem is complicated by the fact that we must also address load balancing and parallelism.

For the purposes of this paper, we restrict the problem to merely identifying the best mapping of threads to nodes, given a constant and equal number of threads on each node. We investigated several ways of identifying good mappings. We used integer programming software to identify optimal mappings. We developed several heuristics based on cluster analysis [10], and showed that two heuristics identified thread mappings with cut costs that were within 1% of optimal for all of our applications. We collectively refer to these heuristics as *min-cost*.

However, a much simpler heuristic, *stretch*, appears to perform almost as well on the applications discussed in this paper. *Stretch* consists merely of maintaining the initial thread ordering and attempting to divide the threads equally among the nodes. For example, given a 64-thread application, we would map the threads on to four nodes by putting the first 16 on node 0, the second 16 on node 1, etc. The reason that stretch works well is that the majority of communication in our applications is either nearest-neighbor or approximately all-to-all. In the former case, stretch is exactly the right approach. In the latter, all configurations are equivalent. For evidence of nearest-neighbor sharing patterns, look at any of the correlation maps in Table 3. The all-to-all communication is not as obvious, but can be inferred as well. Consider the correlation maps for Ocean and LU. Both have uniform dark backgrounds (all-to-all sharing), with even darker boxes near the diagonal (nearest-neighbor communication). Since *stretch* and *min-cost* perform so similarly, we only present information for *min-cost* below. Note that *stretch* will often move more threads at migration points than other approaches.

Applications		Time (secs)	Remote Misses	Total Mbytes	Diff Mbytes	Cut Cost
Barnes	m-c	43.0	120730	218.1	29.3	125518
	ran	46.5	124030	254.2	29.3	129729
FFT7	m-c	37.3	22002	172.2	169.2	8960
	ran	68.9	86850	685.9	193.4	14912
LU1k	m-c	7.3	11689	121.3	9.6	31696
	ran	97.1	231117	1136.2	145.2	58576
Ocean	m-c	21.2	123950	446.3	228.7	26662
	ran	28.9	171886	605.5	240.4	29037
Spatial	m-c	240.1	125929	551.8	107.7	273920
	ran	273.7	249389	870.8	115.8	289280
SOR	m-c	3.6	881	5.4	5.0	28
	ran	5.9	8103	47.7	46.0	252
Water	m-c	19.3	20956	49.0	6.9	21451
	ran	21.1	33188	72.0	6.9	23635

Table 6: 8-node performance by heuristic

Table 6 shows communication requirements, counts of remote misses, and overall performance for each application with both *min-cost* ("m-c") and a random assignment ("ran") of threads to nodes.

6. Related work

Thread migration has also been studied in the Millipede [2] and PARSEC [3] DSMs. PARSEC implements sequential consistency [11] rather than one of many high-performance relaxed consistency models. This makes comparisons difficult, as sequentially-consistent systems suffer from both false and true sharing. Relaxed consistency models hide false sharing effectively without recourse to multi-threading [12]. Thread-scheduling algorithms on modern systems, therefore, only address performance problems due to true sharing. Furthermore, the level of false sharing in both systems is higher than a typical sequentially-consistent system, as neither system incorporates a "delta interval" mechanism. This mechanism freezes newly arrived pages for a pre-determined period of time before allowing them to be stolen away. This optimization has long been known to be crucial to the performance of single-writer DSM protocols [13].

Both systems implement forms of passive correlation scheduling, in which remote page faults are used to gain information about data sharing between threads. As discussed in Section 4, this technique fails to provide information about the affinity between local threads, and can cause thread thrashing.

In addition to correlation scheduling, PARSEC also implements a "suspension scheduling" algorithm that temporarily suspends threads involved in page thrashing. Suspension scheduling effectively deals with the same performance problems as the delta mechanism, which is only needed in single-writer protocols. Hence, suspension scheduling is unlikely to be of use with more modern underlying consistency mechanism. This is crucial in evalu-

ating the performance results in this paper, as two of the three applications speed up only through suspension scheduling. The performance of the remaining application, water-nsquared from SPLASH-2 [6], improves by approximately 17%. However, the paper gives no absolute performance information for this application, and in fact does not specify how many nodes are used.

7. Conclusions and future work

This paper has described new methods of obtaining and using *thread correlation* information in multithreaded DSMs. We first showed that thread correlation information can be used to derive *cut costs*, which correlate well with communication costs and can be used to predict overall performance. Hence, cut costs can be used to predict the performance of arbitrary mappings of threads to nodes.

We showed that previous approaches to obtaining thread correlation information may require multiple rounds of migrations to stabilize, and still fail to acquire all relevant information. We described the *active correlation tracking* mechanism, which can be used to acquire complete thread correlation information without migration.

Finally, we described the performance of the CVM for a variety of applications, with two different approaches to creating thread mappings. We argued that simple heuristics can approximate optimal mappings for the applications discussed in this paper.

We plan to extend our results with dynamic applications. Several of the applications that we discuss in this paper are potentially dynamic because they reflect physical processes in which interacting bodies move relative to each other. By design, however, these movements are rarely significant with the default inputs sets of the applications that we are currently using. For the full version of this paper, we will present results showing the impact of thread migration on adaptive, irregular codes [14]. Note that the *stretch* heuristic is only applicable to applications with static sharing patterns. We will need to rely on *min-cost* in order to obtain good performance for adaptive applications.

8. References

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

[2] A. Itzkovitz, A. Schuster, and L. Wolfovich, "Thread Migration and its Applications in Distributed Shared Memory Systems," Technion IIT LPCR #9603, July 1996.

[3] Y. Sudo, S. Suzuki, and S. Shibayama, "Distributed-Thread Scheduling Methods for Reducing Page-Thrashing," in *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, 1997.

[4] D. Scales and K. Gharachorloo, "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory," in *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[5] 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.

[6] 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.

[7] K. Thitikamol and P. Keleher, "Multi-Threading and Remote Latency in Software DSMs," in *The 17th International Conference on Distributed Computing Systems*, May 1997.

[8] 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.

[9] E. Dahlhaus, D. S. Johnson, C. H. Papdimitriou, P. D. Seymour, and M. Yannakakis, "The Complexity of Multiterminal Cuts," *SIAM Journal on Computing*, vol. 23, pp. 864-894, 1994.

[10] R. A. Jarvis and E. A. patrick, "Clustering using a similarity based on shared near neighbors," *IEEE Transactions on Computers*, vol. C-22, November 1973.

[11] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. C-28, pp. 690--691, September 1979.

[12] C. Amza, A. L. Cox, K. Rajamani, and W. Zwaenepoel, "Tradeoffs between False Sharing and Aggregation in Software Distributed Shared Memory," in *Proceedings of the Principles and Practice of Parallel Programming*, 1997.

[13] B. Fleisch and G. Popek, "Mirage: A Coherent Distributed Shared Memory Design," in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[14] H. Han and C.-W. Tseng, "Improving Compiler and Run-Time Support for Adaptive Irregular Codes," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, October 1998.