Impact of Sharing-Based Thread Placement on Multithreaded Architectures

Radhika Thekkath and Susan J. Eggers
Dept. of Computer Science & Engg., University of Washington, Seattle 98195
{radhika,eggers}@cs.washington.edu

Abstract

Multithreaded architectures context switch between instruction streams to hide memory access latency. Although this improves processor utilization, it can increase cache interference and degrade overall performance. One technique to reduce the interconnect traffic is to co-locate threads that share data on the same processor. Multiple threads sharing in the cache should reduce compulsory and invalidation misses, thereby improving execution time.

To test this hypothesis, we compared a variety of thread placement algorithms via trace-driven simulation of fourteen coarse- and medium-grain parallel applications on several multithreaded architectures. Our results contradict the hypothesis. Rather than decreasing, compulsory and invalidation misses remained nearly constant across all placement algorithms, for all processor configurations, even with an infinite cache. That is, sharing-based placement had no (positive) effect on execution time. Instead, load balancing was the critical factor that affected performance. Our results were due to one or both of the following reasons: (1) the sequential and uniform access of shared data by the application's threads and (2) the insignificant number of data references that require interconnect access, relative to the total number of instructions.

1 Introduction

There are two memory-system related reasons for sublinear speedup in multiprocessors. First, data sharing among threads scheduled on different processors can lead to excessive data movement and higher network traffic [2, 8]. Second, large-sized networks have long memory access latencies. Together, they can cause considerable performance degradation.

Multithreaded architectures address the second problem of long latencies by context switching between threads and executing useful instructions while waiting for memory accesses to complete [4, 10, 12, 18]. This decreases processor idle time, i.e., improves processor utilization. However, frequent context switching can exacerbate the first problem by increasing inter-thread conflict misses from the combined working sets of multiple threads. Therefore the improved processor utilization could be offset by a rise in interconnect traffic, the result being poorer overall performance.

Techniques for reducing the traffic include larger caches to better accommodate the multiple working sets, cache set associativity to reduce conflict misses and co-locating threads that share data on the same processor to reduce sharing traffic. Co-locating threads has potential for reducing misses in two ways. First, compulsory misses should decrease, because the first reference to shared data by all threads after the first will hit in the cache. Second, invalidation misses should also drop, because shared data of co-located threads is not affected by invalidations issued from that processor. The common assumption is that the reduction of compulsory and invalidation misses will compensate for the increase in conflict misses caused by the multithreaded working set.

This paper evaluates the effectiveness of sharing-based thread placement, in particular, the hypothesis that co-locating threads that share data on the same processor will decrease compulsory and invalidation misses. To test the hypothesis, we evaluated a variety of thread placement algorithms that explore a wide range of realistic policies based on program metrics. The underlying architecture for the simulations was a multithreaded shared memory multiprocessor, in which we varied the number of hardware contexts per processor. Our workload consisted of fourteen coarse- and medium-grained, explicitly parallel programs that are representative of real applications in the scientific and engineering problem domains. Each placement algorithm was based on metrics derivable from program characteristics, both sharing-related metrics, such as the number of shared references among co-located threads, or other non-sharing related information, such as thread length. Values for the metrics were obtained by statically analyzing program traces of each thread, separately.

Our results contradict the hypothesis. Rather than decreasing, compulsory and invalidation misses remained fairly constant across all placement algorithms, for all processor configurations. Even when simulating with an effectively infinite cache, which exaggerates the effect of compulsory and invalidation misses and converts some conflict misses into invalidation misses, there is no variation in compulsory and invalidation misses across placement algorithms. Thus, placing threads that share data on the same processor had no effect on performance, and was sometimes counterproductive. Instead, thread load balancing was the primary factor that affected performance. The lack of a sharing effect is explained by three key program characteristics: the insignificant amount of interconnect traffic caused by shared data, relative to the total number of memory references; the sequential sharing of data among the threads of the application; and an equal number of shared data references between threads of the application, i.e., uniform data sharing.

The rest of the paper is organized as follows. Section 2 describes the thread placement algorithms used in the study and the rationale for choosing them. Section 3 describes the workload, the statically measured program characteristics and the simulation environment. Section 4 presents and analyzes the results. Section 5 describes related work, and Section 6 summarizes.

2 Thread Placement Algorithms

Given a set of threads and the number of processors to schedule, the goal of a placement algorithm is to map each thread to a specific processor. Threads co-located on the same processor are said to be "clustered" together. Our placement algorithms assume that each thread is initially in its own cluster and then iteratively combine clusters until the number of clusters is equal to the number of processors. The criteria for cluster combining vary across placement algorithms. When it is some measure of inter-thread sharing, the placement algorithm is "sharing-based", otherwise it is not sharing-based. Within the restrictions imposed by a particular cluster-combining algorithm, the threads are equally distributed among the processors; this is called *thread-balancing*, to be distinguished from load balancing, where the *instructions* of the application are distributed equally among the processors.

In the following discussion we will describe the criteria for clustering in each of the placement algorithms and the aspect of cache behavior they are designed to exploit:
(1) SHARE-REFS is the basic sharing algorithm that maximizes shared references on co-located threads. The metric used for clustering is the number of references made by threads in the clusters to their common data addresses. The algorithm is expected to be effective in cases where sharing between different threads varies significantly.
(2) SHARE-ADDR maximizes per processor shared references *per* shared address. For example, given two candidate clusters, each with the same number of shared references, it picks the one with the smaller shared working set, i.e., more references per shared address. It thus makes better use of shared cache blocks, thereby potentially reducing cache conflicts.
(3) MIN-PRIV maximizes shared references (like SHARE-REFS) and, in addition, minimizes the number of non-shared addresses per processor. Hence the dual goals are to reduce conflict misses by decreasing the total number of private cache blocks, as well as to reduce first reference and invalidation misses by co-locating threads that share data.
(4) MIN-INVS minimizes the number of shared references *between* processors that can cause invalidations, rather than maximizing shared references *within* a processor (c.f. SHARE-REFS). During clustering, the algorithm compares the cost of keeping two clusters separated, rather than comparing the savings in combining them. We implemented it as an alternative to SHARE-REFS, since it tries to achieve the same goal, but from a different perspective.
(5) MAX-WRITES maximizes write-shared data references among co-located threads. This is a refinement of the SHARE-REFS algorithm, since it omits read-shared data from the sharing-metric and only includes write-shared data, which is responsible for invalidations. It attempts to statically capture a more accurate picture of invalidation traffic.
(6) MIN-SHARE represents a "worst" case schedule with respect to sharing and is used to bound the performance range induced by sharing effects. It minimizes shared references per processor by co-locating threads which have the *least* shared references.
(7) LOAD-BAL does load balancing based on thread length. The algorithm uses the dynamic length of each thread from the trace, and the resulting placement represents a perfectly load balanced execution. Load balancing is a standard scheduling technique on multiprocessor systems [13, 14, 20]; we use it for performance comparison.
(8) Load balancing (LB) is added to algorithms SHARE-REFS, SHARE-ADDR, MIN-PRIV, MIN-INVS, MAX-WRITES and MIN-SHARE to generate versions of those algorithms that load balance instead of thread-balance when combining clusters. The sharing criterion of the original algorithm is applied first, followed by the load balancing criterion. The load-balancing criterion is deemed satisfied if the total load does not exceed a certain percentage (typically 10%) of the ideal load distribution.
(9) RANDOM is the baseline used for comparison with the other algorithms. It generates a random placement map for the given threads, ensuring thread-balance across processors. This is often what a low-overhead runtime scheduler would adopt, given no a priori application knowledge.

To provide a better understanding of the sharing-based placement algorithms, we discuss the SHARE-REFS algorithm in detail and then illustrate it with a simple example. The other sharing-based placement algorithms differ from SHARE-REFS only in the specific sharing metric they compute, i.e., step 2 of the algorithm.

Recall that the goal of the SHARE-REFS algorithm is, given t threads and p processors, to combine clusters until there are exactly p clusters of the t threads such that shared references are maximized within each cluster. To satisfy the thread-balance criteria, each cluster must have t/p threads if p divides evenly into t; otherwise some processors will have $\lfloor \frac{t}{p} \rfloor$ threads and others will have $\lceil \frac{t}{p} \rceil$ threads.

2.1 The SHARE-REFS Algorithm

Step 1: Let $\mathcal{T}$ be the set of threads ($|\mathcal{T}| = t$) and let $\mathcal{C}$ be a partition of $\mathcal{T}$ into clusters. Let $\mathcal{C}^i$ be the cluster partition at iteration i. $|\mathcal{C}^i|$ is the number of clusters at iteration i. For a cluster $c_a \in \mathcal{C}^i$, $|c_a|$ = number of threads in the cluster c_a.

At the beginning of iteration 1, each thread is in a separate cluster, and $|\mathcal{C}^1| = |\mathcal{T}|$. If $|\mathcal{C}^1| \leq p$, then we are done.

Step 2: Compute the sharing metric between all cluster pairs $c_a, c_b \in \mathcal{C}^i$.

(a) Iteration 1: Initially each cluster has only one thread, so inter-cluster sharing is identical to inter-thread sharing. Therefore, sharing-metric(c_a, c_b) = shared-refs(t_a, t_b), where clusters $c_a, c_b \in \mathcal{C}^1$, $c_a \neq c_b$, and threads $t_a \in c_a$ and $t_b \in c_b$.

(b) For iterations $i > 1$: Consider each cluster pair, $c_a, c_b \in \mathcal{C}^i$ such that $|c_a| > 1$, i.e., it has more than one thread, and $c_a \neq c_b$. Let $\mathcal{A}$ be the set of all shared addresses such that each address in $\mathcal{A}$ is referenced by at least one thread in each cluster c_a and c_b. Then,

$$\text{sharing-metric}(c_a, c_b) = \frac{\sum_{\forall a \in \mathcal{A}} \text{Total refs. to } a}{|c_a| \cdot |c_b|}$$

Note that all shared addresses between threads in the same cluster are ignored since this sharing metric is used solely to decide if it is profitable to combine c_a and c_b. Secondly, to permit comparisons between clusters of unequal sizes, the magnitude of sharing is normalized by the number of thread pairs that contribute to sharing.

Step 3: The clusters with the largest sharing metric value are combined. If the thread-balance criteria will not permit combining, then we pick clusters with the next highest value, and so on, until two clusters can be combined.

Step 4: Repeat steps 2 and 3 until the required number of clusters is formed. When there are more clusters than the number of processors and forward progress is not possible, then the algorithm uses backtracking. It replaces the last combining step by the clusters with the next highest sharing value. The output of the algorithm is a placement map that specifies which threads should be clustered together on a processor.

2.1.1 An Example

For the sake of simplicity, we will assume that sharing between each thread pair is from separate and distinct shared addresses, i.e., no overlap with shared addresses of some other thread pair. Let $t = 5$ and $p = 2$. We need 2 clusters, one with 2 threads and the other with 3 threads.

Iteration 1. in Table 1 shows the sharing metric values for all pairs of clusters. Since threads 2 and 3 (in bold) have the highest sharing value, they are combined to form a single cluster.

Iteration 2. shows the sharing metric values between clusters after the merger in Iteration 1. As an example of applying step 2(b) of the algorithm, we show the calculation of sharing-metric({2,3},{4}). (Recall that there is no overlap in sharing).

$$\frac{\text{shared-refs}(2,4) + \text{shared-refs}(3,4)}{(2 * 1)} = \frac{5+4}{2} = 4.5$$

Iteration 2 combines threads 1 and 5 (in bold), producing the clusters shown in Iteration 3.

Iteration 3. Clusters {1,5} and {4} are combined, and the example is done. The total shared references within each cluster can be found by summing the shared references between all thread pairs in each cluster. Cluster $\{2, 3\}$ totals 10 shared references, and cluster $\{1, 4, 5\}$ has 20 shared references.

3 Methodology

The placement algorithms described in the previous section were evaluated using trace-driven simulation. The simulator modeled a shared-memory multiprocessor whose processors have multiple hardware contexts. Program traces (both data and instruction) from fourteen explicitly parallel applications were generated using the MPtrace [9] parallel tracing tool on a Sequent Symmetry [19]. Certain characteristics of the applications were extracted from the trace files and fed to the placement algorithms, which in turn produced maps associating threads with processors. Both maps and program traces were input to the simulator. The next two subsections discuss in greater detail the application suite and its measured program characteristics, and the simulation environment.

3.1 The Applications and their Characteristics

We analyzed two types of explicitly parallel workloads: coarse-grain programs that include some of the SPLASH benchmarks [17], and medium-grain applications that ran under the Presto parallel programming environment [6, 22]. We use the length and number of threads in an application as a measure of its granularity. Coarse-grain programs have fewer, but longer threads, 6.4 million instructions on the average, but as high as 100 million instructions (Table 2). Threads from the medium-grain suite are shorter, on average 0.8 million instructions each, and more numerous.

The programs span a wide range of application domains. LocusRoute, a commercial quality VLSI standard cell router, Pverify, which compares boolean circuits for functional equivalence and Topopt, which performs topological optimizations on VLSI circuits using simulated-annealing, are CAD tools. Cholesky, which does a Cholesky factorization of a sparse matrix, Vandermonde, which is a sequence

Iteration 1 Cluster pairs	{1}{2}	{1}{3}	{1}{4}	{1}{5}	**{2}{3}**	{2}{4}	{2}{5}	{3}{4}	{3}{5}	{4}{5}
Sharing metric	4	3	6	8	**10**	5	7	4	2	6
Iteration 2 Cluster pairs	{1}{2,3}	{1}{4}	**{1}{5}**	{2,3}{4}	{2,3}{5}	{4}{5}				
Sharing metric	3.5	6	**8**	4.5	4.5	6				
Iteration 3 Cluster pairs	**{1,5}{4}**	{1,5}{2,3}	{2,3}{4}							
Sharing metric	**6**	4	4.5							

Table 1: Sharing algorithm example.

Coarse-grain Applications	**Number of Threads**	**Mean Thread Length (in millions)**	**Application Domain**	**Medium-grain Applications**	**Number of Threads**	**Mean Thread Length (in millions)**	**Application Domain**
Locus.	12	n.a.	CAD	Fullconn	28	0.97	Simulation
Water	12	4.9	Scientific	Grav	22	0.76	Scientific
MP3D	12	9.1	Scientific	Health	30	1.21	Simulation
Cholesky	14	5.8	Math	Patch	77	0.49	Graphics
Barnes-Hut	12	5.7	Scientific	Vand.	24	1.82	Math
Pverify	12	101.0	CAD	FFT	32	0.19	Math
Topopt	9	n.a.	CAD	Gauss	127	0.21	Math

Table 2: The application suite.

of matrix operations on a set of matrices, FFT, which does fast Fourier transforms, and Gauss, which does gaussian elimination, are mathematical programs. The scientific applications include Water, which simulates the evolution of a system of water molecules, MP3D, which simulates rarefied hypersonic flow, Barnes-Hut, which simulates the evolution of galaxies using simulated annealing, and Grav, which is a Presto implementation of the Barnes-Hut clustering algorithm. Patch is a graphics application that does radiosity calculations. Fullconn simulates a fully connected set of processors communicating at random, and Health simulates a distributed environment of doctors, patients and health centers.

Programs traces were statically analyzed for characteristics that provided the cluster-combining criteria. A sharing-based placement algorithm that uses information gathered in this way can be approximated by a compiler, using summary side-effect analysis that detects per-thread memory accesses [11]. We did not study placement algorithms that rely on program runtime behavior, such as the order of shared accesses that cause interconnect operations, which would be practically impossible to determine at compile-time.

Table 3 shows values of several of the measured characteristics for both workloads. Inter-thread shared references, based on a per address rather than a per cache line basis[1] is shown for two extremes: thread pairs, or clusters with one thread, and clusters with half the total number of threads in the application (N-way). For all applications (except Topopt), the deviation for N-way mean sharing decreased relative to the pairwise figures. Smaller deviations indicate that sharing-based placement is less important with more threads per processor.

References per shared address reflects temporal locality in the shared address space and is used by SHARE-ADDR. Percentage of shared references is the ratio of data references to addresses in the shared address space, averaged over all threads. MIN-PRIV uses this information to minimize the amount of private data per processor. Since most coarse-grain applications have more accesses to the private address space, they should benefit less from sharing-based placement, because of unavoidable conflicts with private data.

Finally, thread length is used by the load balancing algorithms to uniformly distribute the instructions executed per processor. Applications with thread length imbalances, i.e., higher deviations from the mean, should benefit more from load balanced placements.

Considered together, these values indicate that sharing-

[1] The only inaccuracy in using addresses rather than cache lines is the exclusion of false-sharing effects. However, our applications do not have a significant amount of false sharing. Shared data in Topopt and Pverify were statically restructured [11] to eliminate false sharing. After restructuring, false sharing misses were 1.5% (Pverify) and 1.7% (Topopt) of the total data misses with a 32 KByte cache. The remaining programs had very little false sharing in the original source, from as little as 0.2% (Grav) to 5.8% (Water).

Applications	Pairwise shared references (in 1000s)		N-way shared references (in 1000s)		References per shared address per thread		Total references to shared-addr. space by total references	Trace length simulated per thread (in 1000's)	
	Mean	Dev(%)	Mean	Dev(%)	Mean	Dev(%)	(%)	Mean	Dev(%)
LocusRoute	527	14.0	7911	4.6	15	22.5	57.4	1055	14.6
Water	202	13.9	2986	1.6	23	2.8	71.7	467	2.4
MP3D	897	0.8	13473	0.0	24	0.0	82.6	1674	0.9
Cholesky	2008	1.8	42264	0.2	24	3.7	17.1	2994	0.0
Barnes-Hut	349	6.9	5236	5.4	8	0.0	58.6	597	7.0
Pverify	700	14.7	10508	2.7	98	26.7	91.7	1095	22.8
Topopt	1238	9.7	9988	31.5	611	7.3	50.7	2934	0.0
Fullconn	63	88.8	5628	1.2	493	92.6	95.6	974	6.1
Grav	42	47.0	2353	26.1	43	35.4	98.2	763	38.9
Health	71	133.7	6479	39.6	854	189.7	93.5	1208	95.2
Patch	12	32.2	9227	0.8	73	22.1	97.4	488	59.1
Vandermonde	39	242.6	2422	64.7	1647	80.9	98.7	1819	80.3
FFT	3	84.5	346	3.3	42	69.2	72.4	191	187.6
Gauss	52	41.2	105072	2.8	26	10.5	95.0	210	84.6

Table 3: Measured Characteristics

Architectural Parameters	Values
Number of Processors	2–32
Number of Hardware Contexts	1–64
Non-memory instruction	1 cycle
Context Switch Time	6 cycles
Cache Size	32,64KB
Cache Block Size	16 Bytes
Cache Hit Time	1 cycle
Network Latency	50 cycles

Table 4: Architectural Inputs to the Simulator

based placement algorithms should have more impact on medium-grain programs and architectures with fewer hardware contexts per processor.

3.2 Simulation Environment

For each application, the simulator takes as input an architectural description and a placement map. The simulator simulates a distributed shared-memory multiprocessor in which processors are connected by a simple multipath interconnection network and cache coherency is maintained with a distributed, directory-based cache coherency protocol [7]. The simulator comprises three modules: processor, cache and the interconnection network.

Each processor models multiple hardware contexts and a round-robin context switch policy. A context switch takes 6 cycles, the time to drain the execution pipeline. A context switch is initiated either by a cache miss or a synchronization instruction from the currently executing thread. The placement maps, i.e., thread clusters generated from the placement algorithms, specify which threads are to be scheduled on individual processors. This is a static assignment that does not vary during the simulation. The simulator assumes that all threads have been loaded into the hardware contexts. Table 4 is a complete list of the architectural parameters specified to the simulator and the range of values used in our experiments. Values for the number of processors, number of hardware contexts and cache size are changed for each application and desired threads/processor configuration. The processor unit also maintains statistics on the cycles spent doing useful work, context switching and idling.

The cache unit models a direct-mapped cache with a hit time of 1 cycle. Restricted by the practical limit on trace lengths, we scaled down both the data set size and the cache size [15] to maintain a realistic ratio between the two. Health and the coarse-grain programs use a 32 KByte cache and the other medium-grain programs use a 64 KByte cache. The cache unit maintains separate statistics on the individual cache miss components of compulsory, intra-thread conflict, inter-thread conflict and invalidation misses. We also assume that all instruction references take 1 cycle, i.e., they always hit in a separate I-cache.

We assume a multipath network and do not explicitly model network contention. Instead, we use a latency value of 50 cycles, approximating the average memory latency of a moderately-loaded Alewife-style multiprocessor [3]. We assume a local memory access time of 8 cycles.

We vary the number of processors and hardware contexts to study the impact, if any, on caches that are stressed to different degrees. With only two processors and many threads per processor, conflict misses should be high, while with many processors and fewer threads per context, the cache is effectively larger and will incur fewer conflict misses.

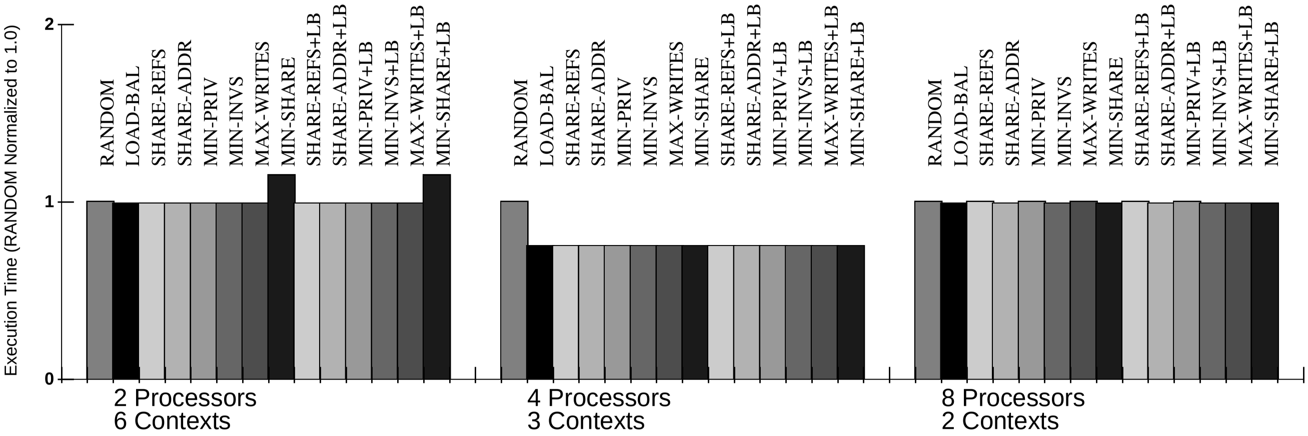

Figure 1: Execution time for LocusRoute. (The height of each bar represents, for a specific placement algorithm, the maximum execution time over all the processors, normalized to RANDOM. The X-axis varies the number of processors and hardware contexts within each processor.)

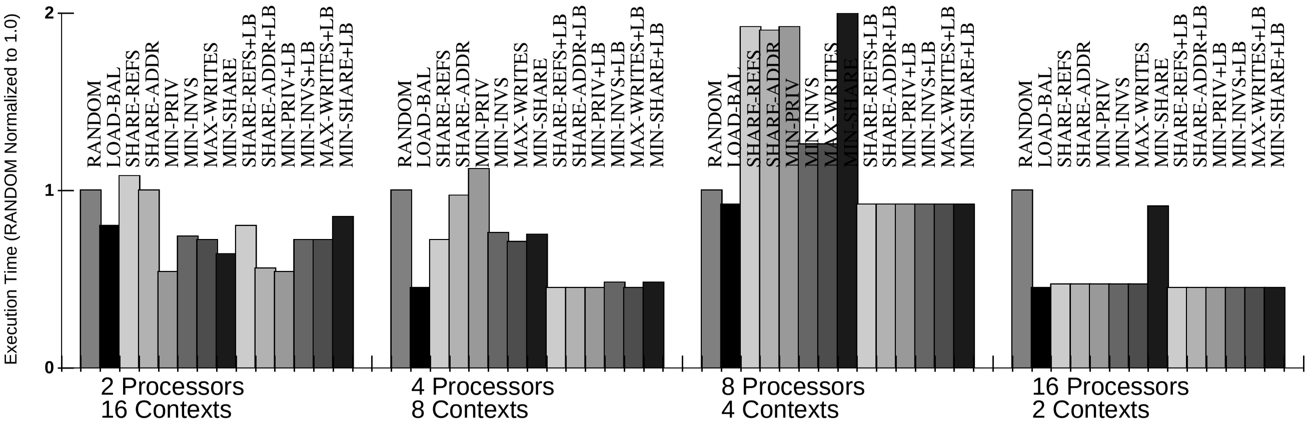

Figure 2: Execution time for FFT (normalized to RANDOM)

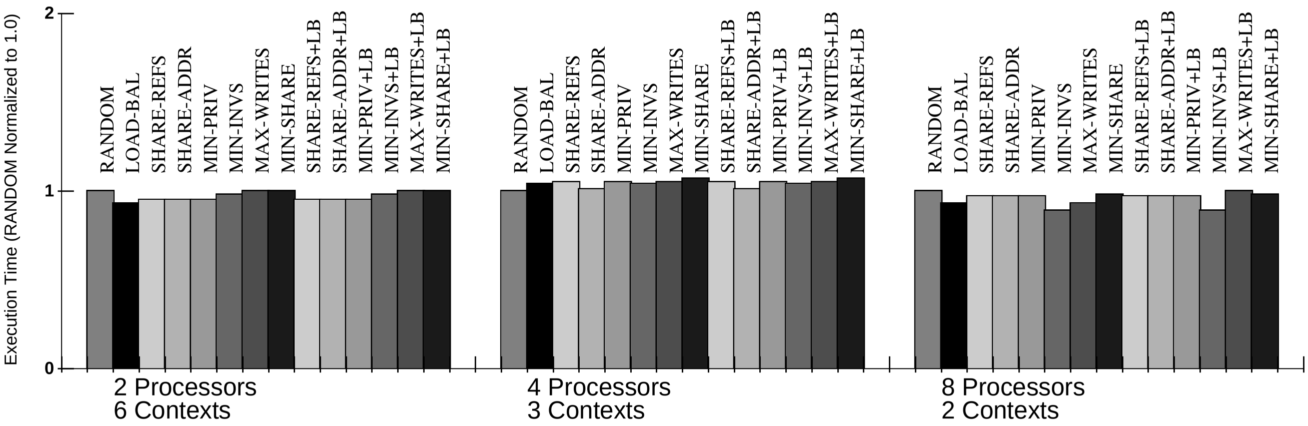

Figure 3: Execution time for Barnes-Hut (normalized to RANDOM)

4 Experimental Results

In this section, we first discuss the results from simulating a wide variety of placement algorithms on different processor configurations and provide an explanation for the obtained results. We then report results from using infinite caches. Since capacity and conflict misses are eliminated in infinite caches, these results represent upper bounds on the benefit of using sharing-based placement algorithms.

4.1 Thread Placement Algorithms

Our experiments show that load balancing is the key factor affecting execution time, for all applications and the majority of threads/processor configurations. Load-balancing has the greatest effect where there is a large deviation from the mean thread length and few threads per processor (i.e., many processors). This is because the potential for imbalance is greater in these cases. With non-uniform thread length and smaller number of processors, load balancing often produces a placement that has far fewer threads on one processor than the others. When locks and other synchronizing instructions block many threads, the processor with fewer scheduled threads loses its ability to hide long latencies. It's idling results in higher execution time for the whole program. On the other hand, with a uniform thread length and/or many threads per processor, a random placement has a good chance to be load balanced.

We now examine the applications and placement algorithms in greater detail. LOAD-BAL very rarely performed worse than RANDOM or any of the sharing-based placement algorithms when the applications had thread length deviations of 15% or greater. (In the rare case when RANDOM did better than LOAD-BAL, the execution time was smaller by only 1.6%). Within this group, LocusRoute (Figure 1) has the lowest thread length deviation for which load balancing was a factor; LOAD-BAL executed 25% faster than RANDOM with four processors. FFT (Figure 2) has the largest deviation of any application (187.6%); comparable figures for it were 8% to 55%. Although its thread length deviation was large (85%), load balancing had a smaller impact for Gauss, since it has a large number of threads (127—the largest of any application).

When the thread length deviation was smaller, 7% or less, LOAD-BAL was comparable to RANDOM. Barnes-Hut (Figure 3) is typical of these programs, where none of the placement algorithms do appreciably better than any other.

Contrary to the current hypothesis, the sharing-based placement algorithms did not lower execution time. Even for those applications with relatively large deviations in pairwise sharing—Locusroute, Pverify and Topopt in the coarse-grain suite and all the medium-grain applications—sharing-based placement did worse than LOAD-BAL. This was not only true for the two threads per processor configuration, where pairwise sharing is applicable, but also for processors with more hardware contexts. On machines with the most threads per processor (only two processors), the sharing-based placement algorithms sometimes did as well as, or better than LOAD-BAL, e.g., MIN-PRIV with FFT. However, this had more to do with the positive effect of thread balancing in the presence of locks and barriers than with sharing.

Several seemingly anomalous results can be explained by the predominant influence of load balancing. For example, some sharing-cum-load balancing algorithms do worse than LOAD-BAL, e.g., MAX-WRITES+LB on Grav (not shown) with eight processors. In these cases, the placement algorithms attempted to satisfy their sharing criterion first, and thus compromised on the load balancing requirement. With LOAD-BAL, the load across processors varied by 15% for Grav. while MAX-WRITES+LB had a 36% variation in load and therefore took twice as long to execute.

In a few rare situations, e.g., Patch with sixteen processors and LOAD-BAL, we observed thrashing when two co-located threads frequently conflicted for the same cache block. This phenomenon has also been reported by Agarwal [1]. The thrashing processor had an order of magnitude more inter-thread conflict misses than other processors, and therefore took longer to complete execution. Set associative caching would address this problem.

4.2 Explaining the Insensitivity of Performance to Sharing

Recall that the original hypothesis argued that co-locating threads that share data on the same processor would decrease interconnect traffic by reducing compulsory and invalidation misses. We tested that hypothesis for all applications by examining the components of misses for all placement algorithms, and all threads/processor configurations. Our results refute the hypothesis; compulsory and invalidation misses remained nearly constant in all cases. This was true not only when compulsory misses and invalidations were a small proportion of total misses, on average 10%, but, more importantly, when they were a large component—on average 40%. The cache miss component graph in Figure 4, typical of all applications, shows that the main effects of decreasing the number of threads per processor are a reduction in conflict misses, due to an effectively larger cache, and a shift from inter-thread to intra-thread conflicts, as the number of threads per processor decreases. Some conflict misses become invalidation misses, but the proportions remain invariant across all placement algorithms, for a given threads/processor configuration.

To investigate the insensitivity of the compulsory and invalidation misses to the placement algorithms, we measured the amount of coherency traffic (invalidation misses and invalidations) and compulsory misses generated at runtime. Dynamic measurements provide statistics on actual sharing traffic, unlike a static, per-thread examination of the

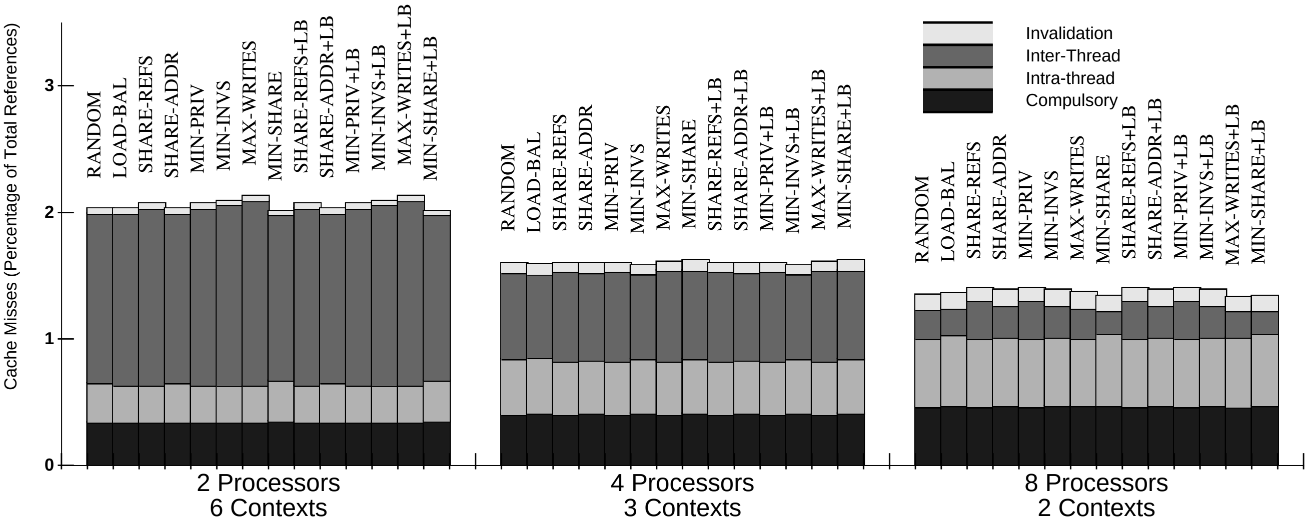

Figure 4: Cache miss components for LocusRoute

Applications	**Pairwise static shared references (in 1000s)**		**Pairwise coherence traffic & compulsory misses (in 1000s)**		**Static shared references (% of total references)**		**Coherence traffic and compulsory misses (% of total references)**	
	Mean	**Dev(%)**	**Mean**	**Dev(%)**	**Mean**	**Dev(%)**	**Mean**	**Dev(%)**
LocusRoute	527	14.0	4.21	24.0	66.6	12.6	0.53	24.2
Water	202	13.9	3.40	59.5	61.4	13.5	1.03	59.5
MP3D	897	0.8	45.64	2.2	64.5	0.4	3.28	2.4
Cholesky	2008	1.8	6.52	20.4	86.6	1.8	0.28	20.4
Barnes-Hut	349	6.9	4.48	6.3	69.1	4.5	0.89	5.2
Pverify	700	14.7	3.83	37.5	74.2	1.7	0.41	35.1
Topopt	1238	9.7	1.60	80.6	87.8	19.1	0.10	63.9
Fullconn	63	88.8	0.94	59.5	10.4	91.4	0.15	60.1
Grav	42	47.0	1.99	36.6	8.3	37.2	0.41	38.4
Health	71	133.7	0.34	140.8	10.8	223.0	0.11	370.2
Patch	12	32.2	0.16	67.8	4.7	25.2	0.06	59.9
Vandermonde	39	242.6	0.03	117.9	7.0	215.7	0.01	386.7
FFT	3	84.5	0.17	143.5	3.3	57.3	0.21	118.5

Table 5: Comparing static shared references with measured coherency traffic.

trace files, which counts references to shared data and does not incorporate any cross-thread temporal information. In order to obtain the maximum amount of coherency traffic between individual pairs of threads, we simulated a system with one thread per processor and as many processors as the number of threads in the application. Our simulations produced three results. First, they explained the lack of any benefit to execution time from the sharing-based placement algorithms. Second, they accounted for the invariance of invalidation and compulsory misses across all placement algorithms. And third, they exposed the pitfalls of using sharing characteristics obtained by statically analyzing single thread traces.

Table 5 shows that the coherency traffic measured between processor pairs (column 4) is one to two orders of magnitude lower than the inter-thread pairwise shared references statically counted from the trace files (column 2). Also, as a percentage of the total references, the compulsory misses and coherence traffic measured at runtime (column 8), was extremely low, ranging from 0.1% to 3.3% for the coarse-grain applications and 0.01% to 0.4% for the medium-grain. This is one to three orders of magnitude lower than the percentage of statically counted shared references in the trace files (column 6).

The insignificant amount of measured coherency traffic, despite the large proportion of shared references in the trace files, is due to two main factors: the sequential nature of inter-thread sharing, i.e., a processor accesses a shared location multiple times before there is contention from another processor, and, fairly uniform inter-thread sharing across the shared address space. This was evidenced by applications whose threads all shared the same data (e.g., Gauss) and those that widely read-shared but wrote locally (e.g., Barnes-Hut). These two factors were primarily responsible for the failure of sharing-based placement algorithms to impact performance.

In addition, total running time was dominated by activities other than actual sharing. The percentage of total cycles due to compulsory and invalidation misses, (a superset of sharing), averaged only 15% over all applications when using the maximum number of processors. Consequently, sharing plays a less important role and thread length becomes the more important criteria, making load balancing threads the primary factor affecting execution time. (The next section describes a study with infinite caches that eliminates unnecessary cache misses, allowing sharing to be the dominant interconnect activity).

The first of the two factors, sequential sharing, can be deduced from the parallel-algorithmic behavior of the programs. In many programs work is partitioned across the main shared data structures, so that each thread works on a separate partition throughout the entire execution. Alternatively, many of the coarse-grain programs use barriers to separate different phases of work; different threads operate on the same piece of data within a phase, and then change between phases. In either case, the programs have been optimized for data locality. For most of the programs the programmer was responsible for the partitioning; in two of them (Pverify and Topopt) locality-enhancing compiler optimizations automatically restructured the shared data to achieve the same effect [11].

For example, Barnes-Hut which does N-body simulation, computes in each time step the net force on a set of particles and updates their position and velocity. The interaction with other particles decreases with distance; hence the algorithm is parallelized by a spatial partitioning of particles into contiguous zones. During the computation phase, thread sharing is predominantly read-sharing of the particle positions and velocities. The actual data structure update occurs at the end of each time step phase, where each process does a local update for its own particles. Thus, the processes read-share data during the long computation phase, and write once at the end of the phase. Since the computation phase for Barnes-Hut is 1.6 million instructions per thread for our input data set, thread length dominates the write-sharing effect.

FFT provides another example. A detailed analysis of the FFT application used in our suite[5], shows that 73% of all shared elements are *migratory*, i.e., they are accessed in long write runs[2]. Other Presto programs have similar sequential access patterns which account for the small amount of runtime coherence traffic.

[2] Write runs are sequences of writes by a single thread.

The large discrepancy between values obtained from statically analyzing (albeit dynamically generated) per-thread traces and those obtained by running simulations on the same traces questions the utility of a thread placement algorithm that is based on dynamically obtained coherency information. Ignoring the question of how such a placement could be determined a priori, we implemented a placement algorithm that used the dynamically measured coherence traffic as the sharing metric. Since it is based on runtime information, it represents the best possible placement that a sharing-based algorithm can produce. However, the results (not shown), were identical to what was seen before: invalidation and compulsory misses were insensitive to the choice of placement algorithm, and execution time for coherency traffic-based placement was comparable to the static strategies.

4.3 The Issue of Cache Size

It is important to understand how sharing-based placement algorithms will impact performance if very large caches are used. With an infinite cache, capacity and conflict misses are eliminated and some conflict misses become invalidation misses[8]; thus, coherency operations dominate interconnect traffic. We ran a set of experiments to study this issue. We compared the load balanced version (LOAD-BAL) with the best (static) sharing-based algorithm for each application and the (dynamic) coherency traffic algorithm. We approximated infinite caches with 8 MByte caches, sufficiently large to eliminate all capacity and conflict misses from the applications. Three applications each were chosen from the coarse- and medium-grain groups that had the least uniform sharing across threads, and therefore, the most potential benefit from sharing-based placement. Specifically, they had the largest absolute deviation in the measured coherence traffic and compulsory misses for their group[3].

The results indicate that the effects of an "infinite" cache do not significantly improve the performance of sharing-based placement algorithms, relative to load balancing. Table 6 shows the execution times of the best sharing-based and coherence traffic-based algorithms, normalized to LOAD-BAL. The key observation from the table is that the best sharing-based placement algorithm has similar execution times to the load balanced placement algorithm for all configurations. In the few highlighted cases shown in bold, sharing-based algorithms do better, but by 2% at most. LOAD-BAL always performs nearly as well as the

[3] Memory references that resulted in coherency and compulsory miss traffic were a very small percentage of total references (Table 5, columns 8 and 9.). Absolute deviation takes into account the size of the mean, and therefore diminishes the effect of a large standard deviation when the mean is small. For example, Vandermonde has a deviation of 386%, a mean of 0.01% and the absolute deviation is only 0.04%. A small mean and small absolute deviation indicates fairly uniform inter-thread sharing, irrespective of the percentage deviation.

Appli-cations	2 processors		4 processors		8 processors		16 processors	
	Best static sharing algorithm	Coherence traffic algorithm	Best static sharing algorithm	Coherence traffic algorithm	Best static sharing algorithm	Coherence traffic algorithm	Best static sharing algorithm	Coherence traffic algorithm
Water	**0.99**	0.99	**0.99**	0.96	**0.98**	0.95	–	–
Locus	**0.98**	1.11	1.03	1.10	**0.99**	0.97	–	–
Pverify	1.03	1.04	**0.99**	0.97	1.04	1.08	–	–
Grav	1.02	1.03	1.00	1.00	1.00	1.24	1.09	1.00
FFT	1.00	0.63	1.00	1.00	1.00	1.00	1.00	1.00
Health	1.04	1.03	1.04	1.02	1.05	1.02	1.01	1.00

Table 6: Execution times normalized to LOAD-BAL, with an 8 MByte cache (essentially no conflict misses) for the best sharing-based algorithm for each application and a placement algorithm that uses the measured coherence traffic and compulsory misses as the sharing metric. Configurations where static does worse are in bold.

coherency traffic algorithm, and more often than not, it does better. As before, these results are due to program characteristics of sequential and/or uniform inter-thread sharing.

5 Related Work

Past research in the performance of multithreaded architectures has examined the tradeoffs between the number of contexts, the network latency and context switch times. Weber and Gupta [23] estimated, via simulation, the extent to which a multithreaded architecture can overcome the effects of long access latencies. They measured processor efficiency by varying the number of contexts and found substantial improvement. Agarwal [1] presents an analytical performance model that incorporates network traffic, cache interference, context switching overhead and the number of hardware contexts. The paper also has a cache model that takes into account the interference in the cache due to multithreading. The paper's main conclusion vindicated multithreading for a wide range of architectural parameters, provided there existed sufficient network bandwidth. It also showed that high cache miss rates can hurt multithreading performance and that applications with active data sharing improves it. Saavedra-Barrera et al. [16] developed a Markov chain model for multithreaded processor efficiency that uses the number of contexts, the network latency, context switch times and remote reference rate. They also incorporate cache performance degradation in their model. The study shows that few contexts cannot effectively hide very long memory latencies.

Scheduling in *single-context* shared memory multiprocessors has included affinity scheduling in medium-grain, explicitly parallel programs [21], and fine-grain scheduling of loop iterations [13, 14, 20]. Affinity scheduling studies the impact of preferentially running a process on the processor where it previously executed, to take advantage of its already loaded cache state to reduce cache misses. This problem does not apply to the multithreaded situation, where the cache state is shared among short-running processes on multiple hardware contexts.

Fine-grain thread scheduling for multiprocessors has centered around clustering and scheduling loop iterations. The primary focus was to maximize processor utilization via load balancing algorithms that use thread length as the sole criteria. This is a reasonable goal for single-context multiprocessors when processor utilization is an indication of overall performance. Since we are studying multithreaded architectures, where memory latency can be hidden to the processor but not the interconnect, our goal is to achieve better utilization of *both* the processors and the cache.

6 Summary

Our results refute the popular assumption that sharing-based placement algorithms can reduce compulsory and invalidation misses on a multithreaded processor. For all applications in the coarse- and medium-grain workload, both types of misses remained unchanged across all thread placement algorithms—sharing-based and otherwise. The dominant factor affecting execution time was thread length, and indeed we found that load balancing was critical for good performance. Load balancing has the greatest effect where there is a large deviation in thread length and few threads per processor. Load imbalances caused considerable performance degradation.

Two factors contributed to the lack of a sharing effect in our workload, i.e., the insensitivity of performance to sharing-based thread placement. First, the data sharing pattern was uniform across all threads in a program. This was seen both by applications whose threads all shared the same data and those that widely read-shared but wrote locally. Second, programs tended to share data very sequentially. Programmers, or in two cases, the compiler, had optimized shared data structures to improve spatial locality and reduce false sharing. The result was that, although there were a preponderance of shared data accesses (relative to total data references), very few produced interconnect operations, even with an infinite cache.

This means that using statically measured, per-trace metrics from dynamically-generated trace files can lead to erroneous conclusions about the sharing behavior of the program. Static, per-trace counts cannot include cross-processor temporal information that is indicative of the runtime interconnect behavior of the program.

Acknowledgements

We are indebted to John Bennett, Shrinivas Devadas, Ed Felten, Tony Ma, the SPLASH group and David Wagner for letting us use their applications, and to Anant Agarwal, Jean-Loup Baer, Anna Karlin, Hank Levy and Dan Nussbaum for useful comments and discussions. Radhika Thekkath's research is funded by an IBM fellowship and Susan Eggers's by ONR grant #N00014-92-J-1395 and PYI award #MIP-9058439.

References

[1] A. Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[2] A. Agarwal and A. Gupta. Memory-reference characteristics of multiprocessor applications under MACH. *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 215–225, May 1988.

[3] A. Agarwal, B-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. *Proceedings of 17th ISCA*, pages 104–114, May 1990.

[4] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. *International Conference on Supercomputing*, pages 1–6, June 1990.

[5] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. *Proceedings of the 17th ISCA*, pages 125–134, May 1990.

[6] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A system for object-oriented parallel programming. *Software: Practice and Experience*, 18(8):713–732, August 1988.

[7] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. *Proceedings of ASPLOS IV*, pages 224–234, April 1991.

[8] S. J. Eggers and R. H. Katz. The effect of sharing on the cache and bus performance of parallel programs. *Proceedings of ASPLOS III*, pages 257–270, April 1989.

[9] S. J. Eggers, D. R. Keppel, E. J. Koldinger, and H. M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 37–46, May 1990.

[10] R. H. Halstead and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. *Proceedings of 15th ISCA*, pages 443–451, May 1988.

[11] T.E. Jeremiassen and S.J. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. Submitted for publication.

[12] K. Kurihara, D. Chaiken, and A. Agarwal. Latency tolerance through multithreading in large-scale multiprocessing. *International Symposium on Shared Memory Multiprocessing*, pages 91–101, April 1991.

[13] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *Supercomputing '92*, pages 104–113, November 1992.

[14] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[15] E. Rothberg, J.P. Singh, and A. Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. *Proceedings of the 20th ISCA*, pages 14–25, May 1993.

[16] R. H. Saavedra-Barrera, D. E. Culler, and T. von Eicken. Analysis of multithreaded architectures for parallel computing. *2nd ACM SPAA*, pages 169–178, 1990.

[17] J. P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[18] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE, Real-Time Signal Processing IV*, 298:241–248, 1981.

[19] Symmetry Technical Summary. Sequent Computer Systems.

[20] T. H. Tzen and L. M. Ni. Dynamic loop scheduling for shared-memory multiprocessors. *Proceedings of ICPP*, pages II:246–250, August 1991.

[21] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. *Proceedings of the 13th SOSP*, pages 26–40, October 1991.

[22] D. B. Wagner. *Conservative Parallel Discrete-Event Simulation: Principles and Practice*. Ph.D. thesis, University of Washington, Seattle, September 1989.

[23] W-D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. *Proceedings of 16th ISCA*, pages 273–280, June 1989.