

A Framework for Evaluating Software and Hardware Mechanisms for Reducing False Sharing

Stefan G. Berg
Department of Computer Science & Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350 USA
sgberg@cs.washington.edu

October 17, 1996

Abstract

We present a framework for studying the effectiveness of different approaches to reducing false sharing. Our framework includes the implementation of a distributed shared memory multiprocessor simulator that has been designed to work well with several currently existing cache coherency protocols known to reduce false sharing. A suite of seven benchmarks has been ported. These benchmarks exist in two versions, one of them optimized by Jeremiassen's compiler analysis for reducing false sharing. We show the results of running these benchmarks on the simulated architecture and give some possible reasons for why the framework in its current form is not suitable for the study we had designed it for.

1 Introduction

False sharing occurs on virtually all shared memory multiprocessors running real-life applications. It is caused by a mismatch between the layout of shared data in memory and the memory access pattern of the processors. For instance, two processors writing to two different shared variables located inside the same cache block will exhibit false sharing. Even though no data is shared

between the processors, they incur the same cost as true sharing would have. While true sharing is an essential part of a program, false sharing is not and can always be reduced (if not completely eliminated) by changing the data layout or explicitly detecting it at run-time [12].

False sharing can be a sizable impediment to decent parallel performance. One study using coarse-grained applications showed that with block sizes from 8 bytes to 256 bytes false sharing made up 40% to 90% of all cache misses [8]. The misses increase traffic on the interconnect and generally increase the execution time. False sharing tends to be a more serious problem for large cache block sizes and when many processors are present. Due to that, false sharing is also a limiting factor for the scalability of parallel programs.

Not surprisingly attempts have been made to reduce the number of false sharing misses. Some do so by changing the hardware, others by changing the software. The hardware solutions all introduce new cache coherency protocols which try to prevent false sharing misses. For example, the Partial Block Invalidation protocol [5] uses multiple coherency blocks per cache block to reduce false sharing. By using a release consistent memory model, the Send and Receive Delayed [7]

protocol delays invalidations, therefore combining false sharing operations. An invalidation received can be delayed until the next lock instruction; an invalidation issued can be held back until the next unlock instruction. The obvious drawback of these methods is the need for new hardware. Furthermore, in the Send and Receive Delayed protocol, the programmer must use the less natural release consistent memory model when writing programs.

The software approaches either restructure the shared data or change the code to alter the cross-processor memory pattern. One such approach optimizes the layout of the program data structures based on a compile-time analysis [13]. The analysis computes an approximation of the memory access pattern for each processor, tries to pinpoint the data susceptible to false sharing, and suggests a data transformation to reduce it. The appeal of this method is that it works with no hardware modifications in a sequentially consistent memory model.

An unanswered question is whether the approximate software solution can compete with the dynamic hardware solutions. This question cannot be answered trivially, because the results from both hardware and software solutions are promising and very similar. The comparison is interesting because it will tell us if we can get the same or nearly the same speedup and reductions in false sharing misses by recompiling our applications as we can by redesigning the hardware. The purpose of the work described in this paper is to design and verify a framework for doing such a study.

The framework is based on Mint [18], a program-driven memory reference generator that permits easy building and modification of a target system simulator (back-end). We used Loki [11] as a starting point for the target system simulator. It simulates a Power-PC architecture using a split-transaction bus for communication between the processors. Loki implements Spur [14], a bus-snooping cache protocol.

Several changes had to be made to the simulated base to allow for an interesting and unbiased

comparison between the hardware and software schemes. The first major change was to replace the bus with a 2-dimensional mesh of interconnected processors. This was necessary, because a bus-based architecture typically does not support more than four processors [10, 3]. False sharing, however, often does not become noticeable until four or more processors are used.

Second, the bus-snooping cache protocol was replaced by a distributed directory-based cache coherency protocol. Both hardware schemes for reducing false sharing were designed for directory-based protocols; therefore when comparing them to the compiler-based alternative, the latter must use a coherency mechanism in the same family. We decided to implement the Censier & Feautrier directory-based protocol [4], because it is a simple protocol with no false sharing support and it can serve well as a comparison basis to the different hardware false sharing solutions.

To allow easy verification of the simulator, a global monitor was written to detect inconsistencies in the caches. The monitor inspects modified cache lines after each CPU operation. Any inconsistencies between the directory and cache contents cause an error that can be detected and therefore corrected as soon as it occurs.

A workload was put together that will be needed for the actual comparison study. The workload was also instrumental in verifying the simulator and ensuring that the compiler optimizations work in our environment¹. The benchmarks used are Fmm, Radiosity, Raytrace, Pverify, Maxflow, Topopt, and Water. All of these benchmarks had to be ported to work under the given simulator. For each benchmark, an unoptimized version (a programmer-optimized version for Water) and compiler-optimized version

¹The original study on compiler-directed data restructuring was performed on a KSR-2. The KSR-2 is quite different from the multiprocessor simulated by our framework. It has no memory, but 32MB second-level caches on each processor. Processors are located 32 to a base level ring with up to 32 base level rings on a second-level ring.

was ported.

The rest of this paper is organized as follows. First, we will describe in detail the three approaches to reducing false sharing. Then we will describe the implementation aspects of the framework, consisting of the directory-based cache coherency protocol, the interconnect, the global monitor, and the benchmarks. Section four will describe how the simulator was verified and discuss the feasibility of applying the compiler optimizations in this framework. We will conclude the paper in section five.

2 Solutions to False Sharing

2.1 Partial Block Invalidation

The Partial Block Invalidation protocol [5] reduces false sharing by dividing each block into multiple coherency blocks and associating a valid bit with each. On each processor update, only the coherency block that contains the updated word is invalidated in other processors. Therefore false sharing between coherency blocks is eliminated; the amount of *total* false sharing eliminated depends on the size of the coherency block.

Chen and Dubois found that the Partial Block Invalidation protocol reduced miss rates by 13% to 91% for a block size of 128 bytes and coherency block sizes between 16 and 64 bytes. In all cases virtually all of the miss rate reduction was attributable to the reduction in the number of false sharing misses. Smaller coherency block sizes always did better than larger ones.

The costs of this approach are additional state bits for the valid bit for each coherency block and more complicated control logic because more states have been added to the cache.

2.2 Send and Receive Delayed

The Send and Receive Delayed protocol [7] reduces false sharing by delaying invalidations as long as possible. Delaying invalidations helps especially well in the common case where two pro-

cessors invalidate each other's cache line in rapid alternation. If a cache line is repeatedly invalidated, only a single invalidation is sent out when the delay period ends.

Delaying invalidations is made possible by assuming a release consistency memory model. In this model nothing is guaranteed about memory consistency unless memory accesses are enclosed within critical sections by using explicit lock and unlock instructions.

Delays can be applied at the sending and receiving end of invalidations. An invalidation received at a cache can be delayed until its processor executes the next lock instruction. A new cache state, Stale, is used to remember which cache lines have received an unfulfilled invalidation. When a processor executes a lock instruction, all Stale cache blocks are invalidated.

Rather than sending invalidations, a processor holds them back in a small buffer. When the processor executes an unlock instruction, its invalidation buffer is flushed.

Dubois et al. have shown miss rate reductions between 8% and 94% for a block size of 128 bytes when comparing the Send and Receive Delayed protocol to a regular non-delayed protocol [7]. The best results were seen in a picture interpolation program. This program uses virtually no synchronization. Processes pass over the pixel values, reading each, computing the unknowns, and writing the value back. With the lack of synchronization, invalidations can be delayed an extremely long time, making this protocol very effective.

The cost of the Send and Receive Delayed protocol is additional hardware for holding the invalidation buffer and a more complex control logic for keeping track of the new state that was introduced. The protocol must also implement a sequentially consistent cache coherency protocol to be applied to all synchronization variables. (Synchronization variables are the mechanism for implementing a release consistency memory model; therefore accesses to them cannot be delayed.) Finally, if a program requires a sequentially con-

sistent memory model, it will need to be modified (lock and unlock instructions must be added) to work with this protocol.

2.3 Compiler Optimization

The compiler approach reduces false sharing by recognizing and reducing the mismatch between the data layout and the memory accesses of the processors. As we mentioned earlier, this mismatch is the sole cause of false sharing misses. Some of this mismatch can be corrected by restructuring the program data. The data layout is known by simply examining the shared data declarations, but the memory accesses of each processor require an in-depth analysis of the program.

The analysis to compute per-process memory accesses consists of three stages [13]. First, an interprocedural control-flow analysis determines which sections of code are executed by each process. The control-flow graph is annotated accordingly and processes with identical annotation patterns in the control-flow are grouped into process families. In the second stage, an interprocedural non-concurrency analysis examines barrier synchronization patterns within the program to find a control-flow between phases that are guaranteed not to be executed concurrently [9]. The last stage performs an enhanced interprocedural, flow-insensitive, summary side-effect analysis and static profiling on all combinations of process families and phases.

Both stages one and three give information about per-process references to shared data. Stage one identifies reference patterns due to differences in the control-flow. Stage three finds reference patterns due to implicit partitioning of arrays across processes. Stage three first splits arrays into bounded regions based on bounded regular section descriptors. If a descriptor is dependent on a variable which is different across the processes and the descriptor splits the array into disjoint sections, an implicit partitioning of the array across the processes has been found.

The static profiling weights the side-effects with respect to an estimated execution frequency to determine those data structures that have a good chance of being falsely shared.

The heuristic to restructure shared data consists of three transformations. The first two group together data that is mostly accessed by one processor. *Group & transpose* groups statically allocated vectors in which adjacent elements are accessed by different processors and transposes them. *Indirection* can be used when data size and location aren't known at compile time, making the previous transformation impossible to apply. Data is relocated to per-processor heap buffers; the original data location points to the exact location within the heap. The third transformation moves write-shared data and synchronization variables into separate cache lines by padding and aligning the data to cache line boundaries.

The compiler approach reduces between 4% and 85% of the cache misses for coherency block sizes of 16 and 128 bytes. The speedups often show little improvement for few number of processors, but the maximum speedups reach values up to three times that of the unoptimized program [12].

The drawback of the compiler optimization is that it bases its decision to restructure shared data on an approximation of dynamic cross-processor memory accesses. In addition, spatial locality may be reduced in some instances, where data is moved apart. This could increase cache misses and decrease the efficiency of the cache.

3 Implementation

This section will describe in detail the implementation of the framework. We use Mint [18] as the memory reference generator (front-end), because it uses executables as its input (instead of traces), and it has an easy interface to the target system simulator (back-end). We use a modified version of Loki [11] as our back-end. Loki simulates a shared memory multiprocessor with split I & D first-level caches and a unified copy-back second-

level cache. The first-level cache is a write-through cache with a write-buffer for the data section of the cache. Almost everything from block size to memory latency is configurable through the use of command line arguments. We use a 64 KB split, direct-mapped, first-level cache, a 2 MB unified, direct-mapped, second-level cache with 6 cycle minimum latency, and a write buffer capable of holding four words. Memory is distributed across processors in 4 KByte pages. The simulated cache coherency protocol is Spur [14], a bus-snooping protocol. Loki simulates a split-transaction bus and has support for counting false sharing misses.

To support the study, several features of Loki had to be modified, resulting in a rewrite of about 30% of the back-end. Next we will describe these changes, which include a switch to a multipath interconnect and a distributed directory cache coherency protocol, and the addition of a global monitor. The interconnect and cache protocol where necessary to increase the number of processors supported by the hardware to levels which exhibit noticeable amounts of false sharing. In the last section we will describe the work involved in porting the benchmarks. The benchmarks make up the workload for the comparison study and were used to verify the framework for correctness.

3.1 Interconnect

Because busses are usually not capable of sustaining the traffic of more than four modern processors, we replaced the split-transaction bus with a 2-dimensional, bidirectional interconnect. The importance of being able to simulate many processors is that false sharing does not really become noticeable until four or more processors are used. With the multipath interconnect we have a typical platform for running parallel applications which scales to enough processors that false sharing becomes an important issue.

Another reason for implementing a multipath interconnect is that the Send and Receive Delayed protocol and Partial Block Invalidation protocol

are specified in terms of a directory-based protocol. It is unusual to use a directory-based protocol on a bus, because the broadcast capability of the bus is not taken advantage of (as is the case with bus-snooping protocols).

The actual simulation of an interconnect is quite involved, because every node in the interconnect must be simulated down to the routing algorithm. While this method gives exact results, it is resource intensive, programmer intensive, and error prone. Instead we chose to use an analytical model described and validated by Agarwal [1]. It is computationally very inexpensive and gives good results. The model uses the probabilities for routing packets in a switch to compute an expected value and a variance for the number of packets that leave a switch during some cycle. That gives way to a simple formula for computing the transit time for a message.

The parameters of the model include the width of the wires connection the switches, the dimensionality of the network, the number of nodes, and the injection rate (related to the miss rate). The model can also be modified for several physical characteristics. We chose a two-dimensional and bi-directional interconnect without end-around connections. This seemed to be a typical configuration for several multiprocessors (Intel Paragon, Intel Teraflot, Dash).

Data packets incur an additional 60 cycle delay due to the memory latency. All packets are assumed to have a 64 bit header. For acknowledgements or other packets containing no data, there is no additional size requirement. The network width is assumed to be 16 bytes.

We introduced a small optimization at the incoming queue of each directory node. What we noticed is that heavily accessed synchronization variables could cause a lot of traffic on the interconnect. Since the queue at each directory is capable of holding several incoming packets, a situation could arise in which the queue was filled with lock requests. If the matching unlock request arrived at such a point, it would have to wait for all the lock requests to be denied. Our optimiza-

tion was to prioritize the queue and move unlock requests ahead of any other requests.

3.2 Censier & Feautrier

With the replacement of the bus with a multipath interconnect, the bus-snooping protocol had to go. We decided to implement the Censier & Feautrier cache protocol [4, 5], because it is a simple, traditional directory protocol. As such, it can serve as a baseline against which the cache protocols that reduce false sharing can be compared and on which compiler-restructured benchmarks can be used.

The simulator closely models the behavior of the cache protocol. Read and write hits in the first and second level caches are always serviced immediately and do not require any interaction by the coherency protocol. A miss in the second level cache causes a sequence of operations to be performed by at least the cache controller servicing the miss and the directory responsible for the needed memory block. Mint provides a mechanism for scheduling operations (called “tasks”) at arbitrary times in the execution flow of the simulated program. Using this mechanism, the operations are accurately scheduled according to the Censier & Feautrier cache protocol at times determined by the analytical interconnect model.

The sequence of operations initiated after a miss in the second level cache must naturally start at the cache controller that caused the miss. A table lookup based on the current state of the cache line and the access method (read or write) will indicate which request to send to the directory holding the needed memory block. The destination directory is easily determined, because directories are distributed one to each processor based on the index of the page address² modulo the number of processors present. Finally the analytical interconnect model is used to determine when the request should arrive at the destination directory. The operation can now be scheduled using Mint’s support for tasks.

²We use 4K Byte pages.

When the operation at the destination directory is started, code is executed to determine what action needs to be performed. Again tables are consulted to determine whether cache lines on other processors need to be invalidated or if the request can be processed immediately. Depending on the outcome, further operations are scheduled until the miss has been serviced. For read misses this will always result in the affected processor being able to continue its execution. Write misses are non-blocking and do not stall the processor.

3.3 Global Monitor

Without the monitor the framework would never have been completed. It was the single most important tool for detecting and locating errors in the cache coherency protocol and interconnect. The alternative, visually inspecting printouts of chronologically sorted cache states, is extremely slow, tedious, and error prone.

The monitor is called from the back-end whenever a miss in the second level cache has been handled. It verifies the correctness of the cache line just modified. Since the directory is distributed on a multipath interconnect, multiple operations may be outstanding at any point in time. This adds a certain complexity to the monitor. The monitor requires full information of the cache line being checked from the directory and cache controllers. At the point when the monitor is called, the information in the directory is clearly in a consistent state, since it just finished an operation and has not yet started a new one. Other cache controllers, however, operate independently, and it is possible that a cache line has just been invalidated. This may result in a temporary inconsistency (until the directory has been informed of the invalidation) and needs to be taken into account by the monitor.

The first phase of the monitor is responsible for collecting data from all caches and detecting and compensating for any inconsistencies. For example, if a cache line has just been invalidated, the monitor will include this fact in its checks and

not complain that the directory and cache controller have inconsistent state information. In the second phase the monitor aborts the simulation under the following conditions:

- When the *Modified* bit and *Presence* bits are set concurrently. This would indicate that according to the directory a cache line is both shared and owned.
- When the *Modified* bit and *Presence* bits of the directory do not match the corresponding state bits in all cache controllers.
- When the inclusion property does not hold. If a cache line in the second-level cache is invalid, the first-level cache must also have that cache line marked as invalid (or it does not contain that line at all).
- When the data and instruction cache contain the same block of memory. While compilers do not mix instructions and data, it is possible that errors in the simulator cause memory blocks to be copied to both instruction and data caches.
- When a cache line is in the error state. This may happen if the tables defining the cache protocol are incorrectly specified. A table lookup with illegal indices can result in the cache line entering this special state.
- When the instruction cache contains an owned page. Compiler do not generate self-modifying code, but buggy tables could mark cache lines containing program code as owned instead of shared.

Outside the monitor, there are several assertions placed in the program code to ensure that certain other known relationships hold. Mostly these are highly implementation specific and not very interesting to mention here further. For example, while an owned cache line is modified, it is checked that the cache controller making the modifications is indeed the owner indicated by the directory state bits.

3.4 Benchmarks

Seven benchmarks were selected for inclusion in the framework. Four of the benchmarks are Splash-2 benchmarks [20]. The choice of benchmarks was guided by the availability of compiler-optimized versions of the benchmarks. The benchmarks have different speedup curves and different sensitivities to false sharing. In this section we will describe each benchmark and the work required to port them to run correctly on the framework.

The three Splash-2 benchmarks are Fmm, Radiosity, Raytrace. Fmm simulates a two-dimensional system of bodies over a predefined time period, using the adaptive Fast Multipole Method. Radiosity computes the distribution of light in a scene based on the iterative hierarchical diffuse radiosity method. Raytrace renders a three dimensional teacup using ray tracing. The other benchmarks are Pverify [16], Maxflow [2], Topopt [6], and Water [19]: Pverify reads logical descriptions of two circuits and verifies that they are functionally equivalent; Maxflow computes the maximum flow from the distinguished source to sink in a directed graph with edge capacities; Topopt uses simulated annealing to do topological compaction of MOS circuits. Water is a parallel N-body simulator that evaluates the forces and potentials among water molecules in liquid state. Most benchmarks have speedup peaks at between 4 to 20 processors and will therefore be good targets for testing various false sharing optimizations.

The benchmarks were ported in two stages. The goal of the first stage was to get all benchmarks running on the SGI platform. Since Mint can simulate most native SGI executables, stage two (executing with Mint) only involved some minor changes to the linking process and of course further checks that the benchmarks still produced correct results.

For each benchmarks, an optimized and unoptimized version was needed. It is important to note that the unoptimized Splash-2 versions do

not in all cases represent the original code. In the case of the Splash-2 benchmarks, considerable effort (by the Splash-2 programming team) had gone into optimizing the code, including attempts to minimize false sharing. Since this represents effort which the compiler analysis is trying to eliminate, those transformations which the compiler analysis would suggest were sometimes removed from the benchmarks.

Stage one made use of the original versions of the benchmarks and the optimized and unoptimized versions of the benchmarks used in the work done by Jeremiassen [13]. Jeremiassen’s versions of the benchmarks were all designed to compile and run on the KSR-2 shared memory multiprocessor. For those benchmarks where originals existed that could directly be compiled on the SGI platform (using the `pharmacs` macro package), the original was taken as a starting point and modified according to the differences between the optimized and unoptimized versions of the KSR-2 source code. For all others, the optimized and unoptimized versions of the KSR-2 source code were modified to use the `pharmacs` macro package instead of KSR-2 specific calls. Finally, we verified that newly obtained benchmarks still gave results compatible to the originals.

Stage two involved changes to the macro code and compiler options. The simulator explicitly simulates the synchronization primitives. For the simulator to be able to detect the synchronization primitives, all lock and barrier calls were replaced with stubs. Furthermore the simulator cannot handle shared libraries; therefore the linker was instructed to use static libraries for creation of executables in this stage. Again, testing was done to ensure that the benchmarks still gave correct results.

4 Verification

Verification took place at two levels of the framework. First the benchmarks were verified for correctness by running them directly on an SGI and comparing the output to executions of the origin-

als on a KSR-2. In all cases outputs were identical or very close to identical. Certain minor variations in the benchmark output are to be expected, because some benchmarks use random number generators in their algorithm. For brevity we do not show the exact outputs in this paper.

The second step in verifying the framework involved running compiler-optimized and unoptimized versions of the benchmarks on the Censier & Feautrier protocol. We expected to see similar results as those that were published in the original study [13] of the compiler optimization that was conducted using the Spur cache protocol. The results of this step are discussed next.

4.1 Framework Verification

To demonstrate that the simulator reports believable results, we ran optimized and unoptimized versions of each benchmark and compared the results to those obtained by Jeremiassen’s study on the KSR-2 [13]. While we did not expect the results to be comparable in absolute terms, we expected the relative differences of the optimized and unoptimized version to be comparable.

The first results we obtained with Pverify showed promising numbers for simulations on up to nine processors, but beyond that the compiler-optimized version rapidly degraded in performance until at sixteen processors, the optimized program took noticeably longer to complete. An in-depth analysis showed that the compiler-optimized benchmark ran with so few cache misses, that the performance of the directory servicing one frequently used synchronization variable collapsed. The incoming queue at the directory completely filled with lock and unlock requests, creating excessively long delays during those operations. This problem was exaggerated by our use of spin locks (test and test and set) which can cause a lot of traffic when many processors are trying to obtain a lock. The unoptimized benchmark did not have this problem, because it spent more time waiting for the completion of false sharing misses and therefore no

directory had to handle a high rate of synchronization operations.

The poor performance was corrected by speeding up the interconnect in three areas. First, we widened the data path from 16 to 128 bits. Second, the directories at each node were changed to allow for more parallelism. The rate at which directories can retire operations was increased significantly, especially in the case of consecutive read operations on the same block. This increased the complexity of the simulation somewhat, but still matches what an aggressive hardware implementation would aim for. Finally we allowed unlock operations to jump ahead of other operations in the directory queues as described in section 3.1. This eliminated many lock attempts that were guaranteed to fail if the matching unlock operation is at the tail of the queue. In addition we believe this to be a reasonable requirement to make on the hardware.

The combination of these three improvements eliminated the problems exhibited by Pverify and Radiosity and gave us results comparable to the results obtained on the KSR-2. Since some of the other benchmarks exhibited unexpected performance results, too, we have come to the conclusion that it will probably take a similar approach with those benchmarks as we have taken with Pverify. Next we will show and discuss the results of our benchmark suite.

4.2 Pverify & Radiosity

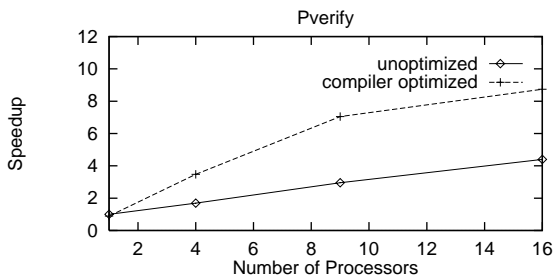


Figure 1: The speedups for optimized and unoptimized versions of Pverify using a 128 byte block size. The baseline of the speedups are the unoptimized results for one processor.

Pverify and Radiosity are the two benchmarks that show promising results on our simulated architecture (fig. 1 & 2). The compiler optimized versions of the benchmarks have considerably higher speedups at all numbers of processors. These speedups are caused by smaller total and false sharing miss rates in all multiprocessor configurations (fig. 3 - 6). For example, at 16 processors, false sharing is reduced by about 95%. While false sharing increases as more processors are added and becomes the dominant source of misses in the unoptimized case, it contributes to only a small part of the misses for the compiler-optimized version. This is compatible with the results Jeremiassen obtained on the KSR.

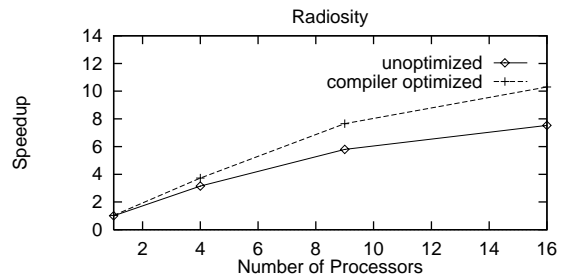


Figure 2: The speedups for optimized and unoptimized versions of Radiosity using a 128 byte block size. The baseline of the speedups are the unoptimized results for one processor.

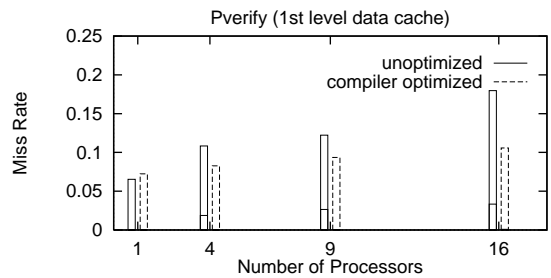


Figure 3: The miss rate of the first level cache showing a breakdown of the false sharing rate (bottom component of each bar). Note that the false sharing rate is occasionally too small to be recognizable on the graph.

On one processor we can typically see a higher miss rate and therefore slightly lower performance for the compiler optimized version. This was the case on the KSR, too. The primary cause for this

is the padding of several data structures and of all synchronization variables. There is no false sharing on one processor. Padding variables therefore only decreases spatial locality and increases replacement misses. Furthermore, the indirection transformation serves no useful purpose, but does introduce additional reference for the indirection pointer. This increases overall references and adds more pressure to the cache.

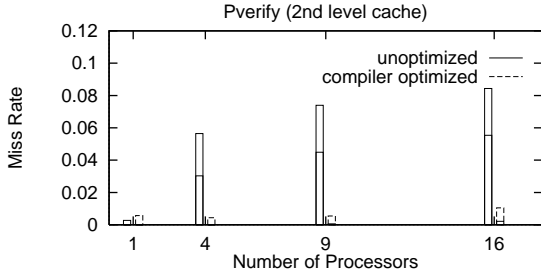


Figure 4: The global miss rate of the second level cache (excluding references and misses of instructions) showing a breakdown of the false sharing rate (bottom component of each bar). Note that the false sharing rate is occasionally too small to be recognizable on the graph.

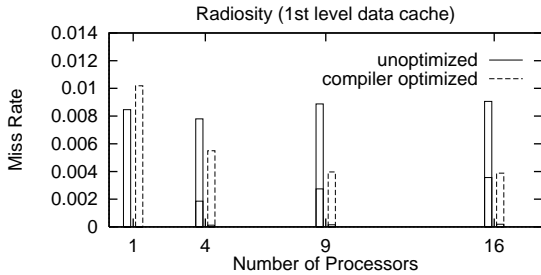


Figure 5: The miss rate of the first level cache showing a breakdown of the false sharing rate (bottom component of each bar).

There are two more characteristics of Pverify’s results worth mentioning here. First we can see a slight reduction in the non-false sharing misses (fig. 3) that can be explained by an improvement in spatial locality due to the indirection transformation. The KSR exhibited this reduction, too. Secondly, compared to the KSR (for which we only have first-level cache data) our results of the first-level cache have a much lower proportion of false sharing misses (fig. 3). The most likely

explanation of this is that the KSR has an eight times larger first-level cache using 2-way associativity (instead of direct mapping) that can hold data longer in its cache, therefore increases the probability of an invalidation followed by a false sharing miss. In the second-level cache, we see a much larger proportion of false sharing misses (fig. 4).

Unlike the KSR results for Radiosity, our results for Radiosity show a reduction in non-false sharing misses similar to the one found with Pverify (fig. 5). While none of Radiosity’s data structures were transformed using indirection, the group & transpose transformation was used on many and can give similar improvements in spatial locality. It is unclear why this effect would be visible on our architecture and not on the KSR.

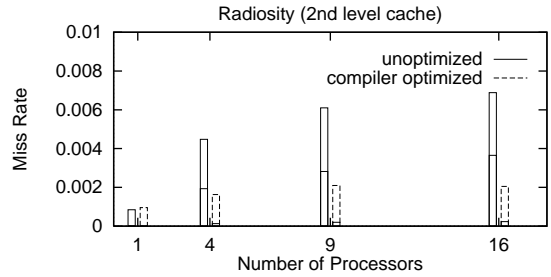


Figure 6: The global miss rate of the second level cache (excluding references and misses of instructions) showing a breakdown of the false sharing rate (bottom component of each bar).

4.3 Fmm, Raytrace & Water

For these three benchmarks we saw no significant difference between the compiler-optimized and unoptimized version (fig. 7 - 15). On the KSR the compiler-optimized version of all these benchmarks did significantly better than the unoptimized version, but typically this did not occur on few number of processors. In the case of Fmm, both versions had identical speedups up to 20 processors. For Raytrace the speedups were identical up to 8 processors. This seems to be caused by relatively few false sharing misses for these three benchmarks. It took significantly

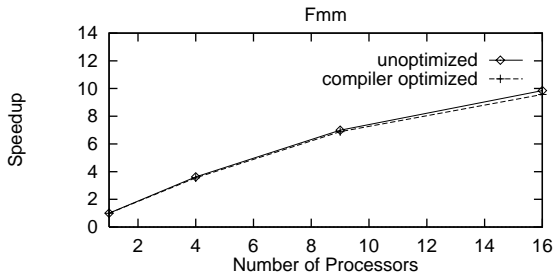


Figure 7: The speedups for optimized and unoptimized versions of Fmm using a 128 byte block size. The baseline of the speedups are the unoptimized results for one processor.

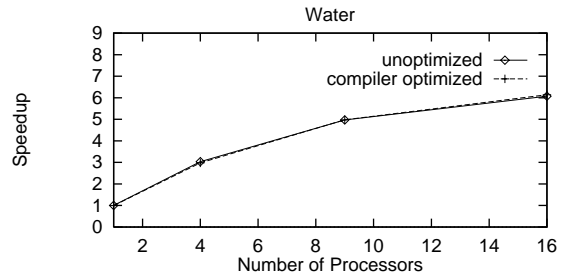


Figure 9: The speedups for optimized and unoptimized versions of Water using a 128 byte block size. The baseline of the speedups are the unoptimized results for one processor.

more processors for the benefit of the compiler optimization to be visible.

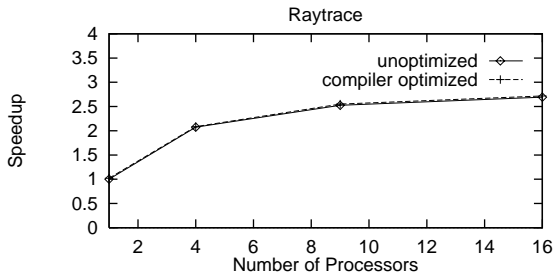


Figure 8: The speedups for optimized and unoptimized versions of Raytrace using a 128 byte block size. The baseline of the speedups are the unoptimized results for one processor.

Even so, the percentage of false sharing misses was reduced in all cases (the graphs often don't show this due to the small number of false sharing misses). We would expect the two versions to show performance differences at some higher number of processors. Due to memory and simulation time constraints we were not able to get these results.

Markedly different are the results for Water. The compiler-optimized version of Water had about 50% higher speedups on the KSR. Our results show no difference even though the small amount of false sharing that does exist was eliminated (fig. 9, 14, 15). All false sharing in Water is due to lock variables that was completely eliminated by the compiler optimization due to padding of synchronization variables. Our results confirm

this. For example, at 16 processors there is a 25% reduction of the busy waiting times at locks. This reduction did not have a noticeable effect on execution time, because a lot of time was spent waiting at barriers (20% of total execution time at 16 processors). One processor interestingly spent very little time at barriers which seems to suggest that the workload was not well distributed between the processors.

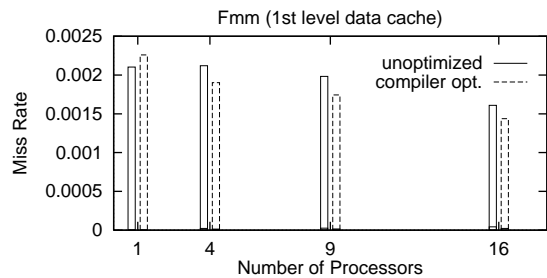


Figure 10: The miss rate of the first level cache showing a breakdown of the false sharing rate (bottom component of each bar). Note that the false sharing rate is occasionally too small to be recognizable on the graph.

4.4 Maxflow & Topopt

This section covers the benchmarks that did not perform well at all. Topopt shows very poor speedups on our architecture, barely surpassing a factor of 2.5 at 9 processors (fig. 17). The KSR speedups are about twice our speedups and at least show good improvements for the optimized version (though only small improvements up to

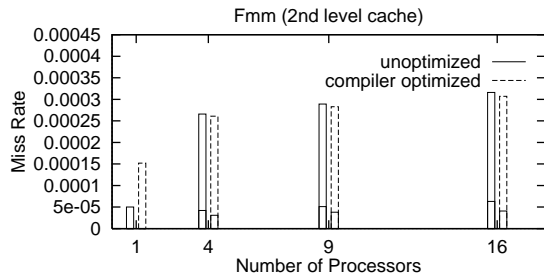


Figure 11: The global miss rate of the second level cache (excluding references and misses of instructions) showing a breakdown of the false sharing rate (bottom component of each bar).

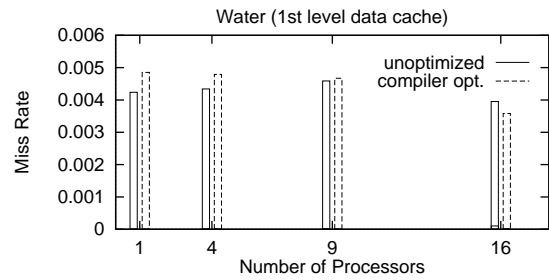


Figure 14: The miss rate of the first level cache showing a breakdown of the false sharing rate (bottom component of each bar). Note that the false sharing rate is occasionally too small to be recognizable on the graph.

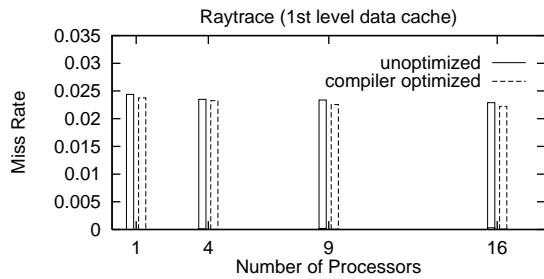


Figure 12: The miss rate of the first level cache showing a breakdown of the false sharing rate (bottom component of each bar). Note that the false sharing rate is occasionally too small to be recognizable on the graph.

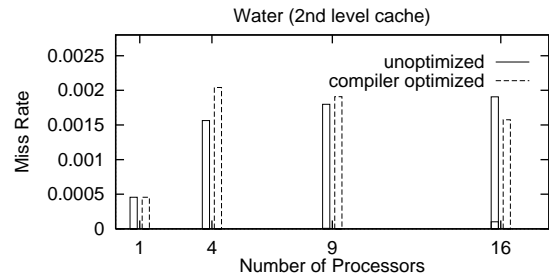


Figure 15: The global miss rate of the second level cache (excluding references and misses of instructions) showing a breakdown of the false sharing rate (bottom component of each bar). Note that the false sharing rate is occasionally too small to be recognizable on the graph.

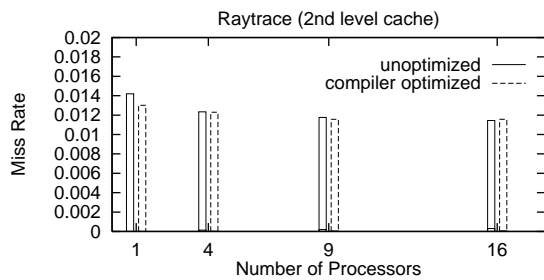


Figure 13: The global miss rate of the second level cache (excluding references and misses of instructions) showing a breakdown of the false sharing rate (bottom component of each bar). Note that the false sharing rate is occasionally too small to be recognizable on the graph.

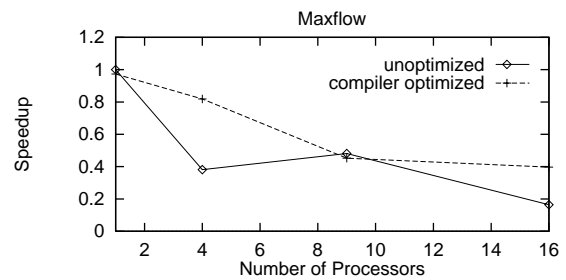


Figure 16: The speedups for optimized and unoptimized versions of Maxflow using a 128 byte block size. The baseline of the speedups are the unoptimized results for one processor.

9 processors which unfortunately was how far we could simulate Topopt due to benchmark problems). Maxflow doesn't run well on our simulator, getting speedups of less than 1 for both versions (fig. 16). On the KSR the unoptimized version achieved a peak speedup of 1.4 at 4 processors after which it, too, dropped to speedups of less than 1. The optimized version on the other hand did much better on the KSR with a peak speedup of 4.3 at 16 processors.

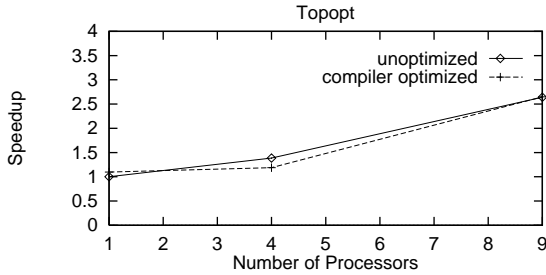


Figure 17: The speedups for optimized and unoptimized versions of Topopt using a 128 byte block size. The baseline of the speedups are the unoptimized results for one processor.

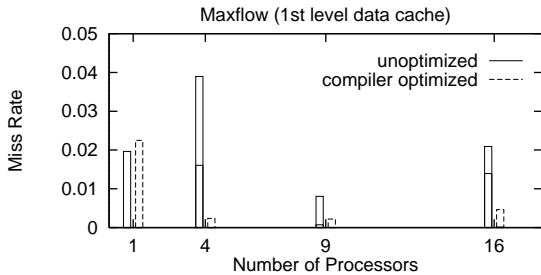


Figure 18: The miss rate of the first level cache showing a breakdown of the false sharing rate (bottom component of each bar). Note that the false sharing rate is occasionally too small to be recognizable on the graph.

Topopt has a considerable load-balancing problem spending 20 to 40% of its execution time waiting at barriers. Since we are not necessarily using the same benchmark inputs as the KSR, we might be experiencing a specific problem with our data set. Waiting at barriers wastes considerable CPU cycles, but should have little relative impact on the miss rates of the two benchmark versions, so let us look at them now. In this category Topopt

does better, eliminating about 50% of all false sharing misses which is close to the results seen on the KSR (fig. 20, 21).

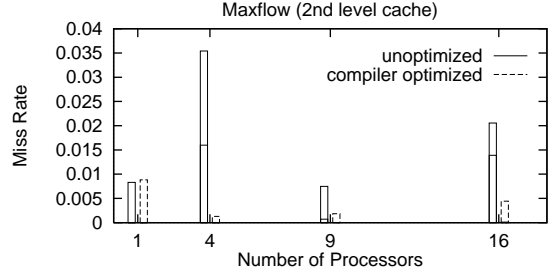


Figure 19: The global miss rate of the second level cache (excluding references and misses of instructions) showing a breakdown of the false sharing rate (bottom component of each bar). Note that the false sharing rate is occasionally too small to be recognizable on the graph.

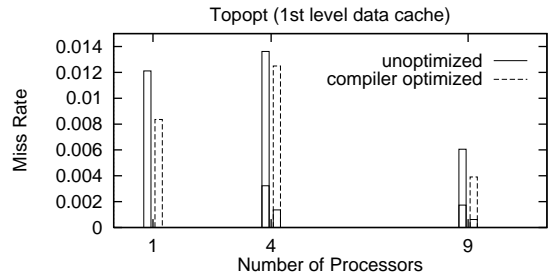


Figure 20: The miss rate of the first level cache showing a breakdown of the false sharing rate (bottom component of each bar).

Maxflow has a similar problem with its barriers, and at larger numbers of processors spends a lot of time trying to acquire locks. The combined time spent at barriers and locks amounts to roughly half the execution time. The KSR did not seem to suffer from these problems. It showed Maxflow as one of the benchmarks with a very high percentage of false sharing misses, most of which were eliminated by the compiler optimization. While this is also the case in our results (fig. 18, 19), it is unclear how to interpret them with our low execution time.

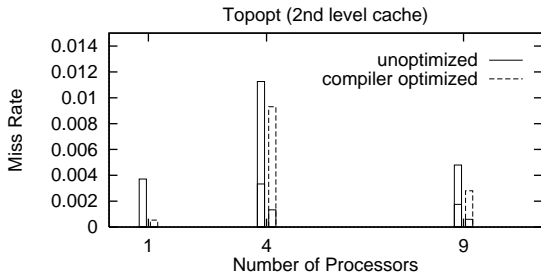


Figure 21: The global miss rate of the second level cache (excluding references and misses of instructions) showing a breakdown of the false sharing rate (bottom component of each bar).

5 Conclusion

This paper described the implementation of a framework for comparing different false sharing solutions. The results we obtained for a number of benchmarks currently do not conclusively show if the compiler optimization works effectively on our architecture. While false sharing misses have been reduced significantly in all cases, only two benchmarks showed major reductions in execution time. In many benchmarks, no significant improvements in speedup were seen, while in a few there were other effects that caused the optimized version to run more slowly.

At least three factors maybe responsible for the discrepancies between these results and those from the previous study done by Jeremiassen.

First, it may be possible that the same in-depth analysis we applied to Pverify must be applied to all other benchmarks to change the architecture until it performs well for a broad spectrum of parallel programs. A number of our benchmarks (Raytrace, Topopt, Maxflow) showed significantly lower speedups than they did on the KSR-2. Two of these benchmarks spent considerable time processing synchronization variables. This could be caused by an inefficiency in the implementation, but there is also a chance that our input data did not distribute well across many number of processors.

Second, the benchmarks need to be simulated with more than 16 processors. Our results for

three benchmarks (Fmm, Raytrace, Water) were inconclusive because of our inability to simulate more than 16 processors. Besides from some minor changes in the simulator code, the main difficulties with simulating more than 16 processors are the memory requirements and simulation times. Our primary platform for running simulations had 64 MB of real memory which was barely enough for some of the benchmarks and forced us to drop to smaller input data for other benchmarks. Execution time was another big factor. In some cases we had to abandon simulations after two weeks of running time and try again with smaller input data. For extended simulation times it may be necessary to implement checkpoints in the simulator to make it possible to restart an aborted simulation. Especially on busy workstations, the chance of an unexpected system shutdown becomes a real problem.

Finally, we have seen several adverse affects of running simulations on our architecture related to the distributed nature of memory. The compiler does not take into account the specific placement of data. The architecture distributes memory across nodes in 4 KByte pages and memory is allocated from these without regard to where it is needed. In most cases the number of memory requests was not evenly distributed across all processors, often some directories serviced an order of magnitude more requests than others. Generally both versions were affected equally and were not making good use of the distributed nature of memory. This issue may need to be addressed first before the issue of false sharing can be looked at. By giving the compiler knowledge of how data is distributed, it should be able to significantly reduce average latency of memory requests. The indirection transformation in particular would be a good target, because it allocates per-process heaps that just need to be allocated on the home node. In other cases it will be more difficult, but not impossible, to determine the best placement of data. New heuristics for the compiler transformations may be required to determine how to transform data and where to place it.

Given these results we must conclude that this framework in its current form is not suitable for a comparison study between software and hardware attempts of reducing false sharing.

6 Acknowledgements

I would like to thank my advisor Susan Eggers for her guidance and support. Her ideas helped me on every aspect of the project. I also want to thank her and Jean-Loup Baer for serving on my quals committee. Finally, I am indebted to Tor Jeremiassen for letting me use Loki as the basis for my work. His support with technical problems related to Loki and the compiler-optimized benchmarks was invaluable to me.

References

- [1] A. Agarwal. "Limits on Interconnection Network Performance." *IEEE Transactions on Parallel and Distributed Systems*. pp. 398-412. October 1991.
- [2] F. J. Carrasco. "A Parallel Maxflow Implementation." *CS411 Project Report*. Stanford University. March 1988.
- [3] M. Cekleov, et al. "SPARCcenter 2000: Multiprocessing for the 90's." *IEEE COMPCON*. pp. 345-53. February 1993.
- [4] L. M. Censier and P. Feautrier. "A New Solution to Coherence Problems in Multicache Systems." *IEEE Transactions on Computers*. Vol C-27. No 12. pp. 1112-8. December 1978.
- [5] Y.S. Chen and M. Dubois. "Cache Protocols with Partial Block Invalidations." *International Parallel Processing Symposium*. April 1993.
- [6] S. Devadas and A. R. Newton. "Topological Optimization of Multiple Level Array Logic." *IEEE Transactions on Computer-Aided Design*. pp. 915-942. November 1987.
- [7] M. Dubois, J. C. Wang, L. A. Barroso, K. Lee, Y.S. Chen. "Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs." *Proceedings of the 1991 Supercomputing Conference*. pp. 197-207, November. 1991.
- [8] S. J. Eggers and T. E. Jeremiassen. "Eliminating False Sharing." *International Conference on Parallel Processing*. pp. 377-81. August 1991.
- [9] S. J. Eggers and T. E. Jeremiassen. "Static Analysis of Barrier Synchronization in Explicitly Parallel Programs." *International Conference on Parallel Architectures and Compilation Techniques*. pp. 171-80. August 1994.
- [10] M. Galles and E. Williams. "Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor." *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*. Volume I: Architecture. pp. 134-43. January 1994.
- [11] T. E. Jeremiassen. "LOKI - A Multiprocessor Cache Simulator." AT&T Bell Laboratories. December 1995.
- [12] T. E. Jeremiassen and S. J. Eggers. "Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations." *Conference on Principals and Practices on Parallel Programming*. pp. 179-88. 1995
- [13] T. E. Jeremiassen. "Using Compile-Time Analysis and Transformations to Reduce False Sharing on Shared-Memory Multiprocessors." PhD Dissertation. 1995.
- [14] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, R. G. Sheldon. "Implementing a Cache Consistency Protocol." *International Symposium on Computer Architecture*. pp. 276-83. June 1985.

- [15] C. P. Kruskal and M. Snir. “The Performance of Multistage Interconnection Networks for Multiprocessors.” *IEEE Transactions on Computers*. Vol. C-32. pp. 1091-8. Dec. 1983.
- [16] H-K. T. Ma, S. Devadas, R-S. Wei, A. Sangiovanni-Vincentelli. “Logic Verification Algorithms and Their Parallel Implementation.” *IEEE Transactions on Computer-Aided Design*. pp. 181-189. February 1989.
- [17] R. Thekkath. “Design and Performance of Multithreaded Architectures.” PhD Dissertation. 1995.
- [18] J. E. Veenstra and R. J. Fowler. “MINT Tutorial and User Manual.” Technical Report 452. August 1994.
- [19] J. P. Singh, W-D. Weber, A. Gupta. “SPLASH: Stanford Parallel Applications for Shared-Memory.” *Stanford Technical Report No. CSL-TR-91-469*.
- [20] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta. “The SPLASH-2 Programs: Characterization and Methodological Considerations.” *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. pp. 24-36. June 1995.