4th Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS), San Diego, CA, Sep. 1993

False Sharing and its Effect on Shared Memory Performance*

William J. Bolosky
bolosky@microsoft.com
Microsoft Research Laboratory
One Microsoft Way, 9S/1049
Redmond, WA 98052-6399

Michael L. Scott
scott@cs.rochester.edu
Computer Science Department
University of Rochester
Rochester, NY 14627-0226

Abstract

False sharing occurs when processors in a shared-memory parallel system make references to different data objects within the same coherence block (cache line or page), thereby inducing "unnecessary" coherence operations. False sharing is widely believed to be a serious problem for parallel program performance, but a precise definition and quantification of the problem has proven to be elusive. We explain why. In the process, we present a variety of possible definitions for false sharing, and discuss the merits and drawbacks of each. Our discussion is based on experience gained during a four-year study of multiprocessor memory architecture and its effect on the behavior of applications in a sixteen-program suite.

Using trace-based simulation, we present experimental evidence to support the claim that false sharing is a serious problem. Unfortunately, we find that the various computationally tractable approaches to quantifying the problem are either heuristic in nature, or fail to agree with intuition.

1 Introduction

A typical (sequentially consistent) shared-memory multiprocessor consists of a number of processors with some form of memory or cache at each processor. In order to increase locality of reference, shared data are generally replicated into the memories or caches of the processors that use them. This replication leads to the problem of data *coherence*—ensuring that all reads of (any copies of) a given datum return the "latest" value. For the purpose of maintaining coherence, memory is grouped into blocks. On a machine with hardware cache coherence, blocks are cache lines; on a machine with VM-based software coherence (i.e., a Non-Uniform Memory Architecture (NUMA) [2, 8, 13] or Distributed Shared Memory (DSM) [16] system), blocks are generally pages. In either case, the coherency protocol does not distinguish among individual words within a block; a write to any word of a block

*This work was supported in part by a DARPA/NASA Fellowship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland, by an IBM summer student internship, by a Joint Agreement for Loan of Equipment (Number 14520052) between IBM and the University of Rochester, by the National Science Foundation under Institutional Infrastructure grant CDA-8822724, and by the Office of Naval Research under contract no. N00014-92-J-1801 (in conjunction with ARPA Research in Information Science and Technology—HPC, Software Science and Technology program).

causes all copies of the entire block to be invalidated or updated. It is therefore possible for references by different processors to disjoint sets of words within a block to result in coherence operations that are not necessary for correct behavior of the program. This is an informal statement of the *false sharing problem*. False sharing has been observed and commented on previously [10, 18], but these studies do not provide sufficiently mathematically precise, convincing definitions. This paper considers several methods for transforming the intuitive idea of false sharing into a precise, usable definition that is able to show false sharing's performance impact on a particular program running on a particular machine.

Section 2 lists criteria for a good definition of false sharing, and then considers several candidate definitions. None is found to be ideal, but several appear to be useful. Section 3 presents the cost component method, which is not a complete definition of false sharing but appears to be a promising direction for future consideration. Section 4 estimates the extent of false sharing for several applications by looking at the overhead and data transfer components of their memory access and coherence costs using trace-driven simulation. The final section summarizes our conclusions.

Practical methods of reducing false sharing are beyond the scope of this paper. The insights offered into the magnitude of the problem, however, indicate that if it could be solved in a general way, it would result in large improvements in parallel program performance, particularly on systems with large block sizes.

An expanded version of this paper appears as chapter 7 of [6].

2 Definitions of False Sharing

Ideally, a definition of false sharing would have the following properties:

- It would **adequately capture** the intuitive notion of false sharing.
- It would be **mathematically precise**.
- It would be **practically applicable**.

To adequately capture the intuitive notion of false sharing, the definition should result in a value that gets bigger as more unrelated things are co-located within blocks, that never grows as blocks are subdivided, and that is zero when the block size is one word. Its value should correspond to the cost savings due to eliminating all of the false sharing in the application in one way or another. That is, to satisfy the intuition criterion, what is defined by a candidate definition should correspond to our informal notion of what constitutes false sharing.

To be mathematically precise, the definition should permit properties of false sharing (such as those in the previous paragraph) to be stated as theorems, and proven. It should present false sharing as a scalar-valued function of a program, some input, and a set of machine parameters. A definition that relies on heuristics will at best provide bounds on false sharing, and at worst inexact approximations that may lie an undetermined distance in either direction from the "truth."

A definition that both captures the intuitive notion and is mathematically precise would in some sense be sufficient. It would be of little practical use, however, it if could not be measured for real programs (e.g. because it required the solution to an NP-hard optimization problem).

The following sections explore some potential definitions of false sharing. All of them fail one or another of the above criteria. They provide insight, however, into the subtlety behind the intuitive concept, and some of them, though imprecise, provide useful approximations to the amount of false sharing in a program. Section 2.1 describes the one-word block definition, which occurs to many people when they are first presented with the idea of false sharing, but which on closer examination badly fails to capture intuition. Section 2.2 describes the interval definition, which uses future knowledge and an optimizing algorithm to quantify false sharing in a precise and intuitively reasonable way. It fails the practicality criterion; there is no known tractable solution to the optimization problem. Section 2.3 addresses the primary weakness of the interval definition by allowing heuristic selection of intervals; it fails the precision criterion. Section 2.4 considers *full-duration* false sharing, and finds that it is too restrictive a definition. Section 2.5 reviews a method used by Eggers and Jeremiassen wherein a program is tuned by hand and measured to determine the extent of false sharing; it is not mathematically precise. The cost-component method is described in its own top level section, section 3. It is a promising but as of yet incomplete technique for measuring false sharing which operates by breaking down the cost of a program execution into its constituent parts. Section 4 uses observations from the cost component method to estimate false sharing in several example programs.

Much of our discussion takes place in the context of a formal model of memory access cost, defined in previous work [5]. The model applies to invalidation-based coherence protocols on sequentially-consistent machines. It captures a program and its input in the form of a shared-memory reference trace, interleaved as the references occurred in practice on some parallel machine (in our studies, an 8-node IBM ACE [12] with uniform access-time memory). The model captures the underlying system in the form of three parameters: the cost of a remote memory reference (infinity in machines that lack this capability), the size of a coherency block, and the cost of copying a block from one location to another, all expressed as multiples of the local cache hit time. In the context of a given system, a coherence *policy* constitutes a mapping from traces to *placements*—time-indexed lists of locations that have copies of various blocks. The cost of a given policy on a given trace is simply the number of memory references for which the data can be found locally, plus the remote reference cost times the number of references for which the data cannot be found locally, plus the block-copy cost times number of times that a new replica is created (the cost of invalidating a copy of a block is assumed to be included in the cost of creating the copy in the first place). Reference [5] presents the cost of several practical policies on a 16-program application suite. It also presents a tractable off-line policy that is provably optimal—that minimizes the total cost of memory accesses and coherence operations.

The optimal policy optimizes the choice between replicating a block and using remote memory operations to access an existing copy. For machines that lack remote reference, there is no choice to be made, and the policy reduces to a straightforward (on-line) invalidation-based protocol. There are two main ideas behind using an optimal policy rather than any particular on-line policy. The first is that it permits changing machine parameters without re-tuning the policy, and the second that it eliminates the possibility of having an on-line policy affect results by favoring one architecture or program over another. An important point in understanding the use of the optimal policy is that previous results [3] show that, at least for a certain class of architectures, straightforward on-line policies can closely approach optimal performance.

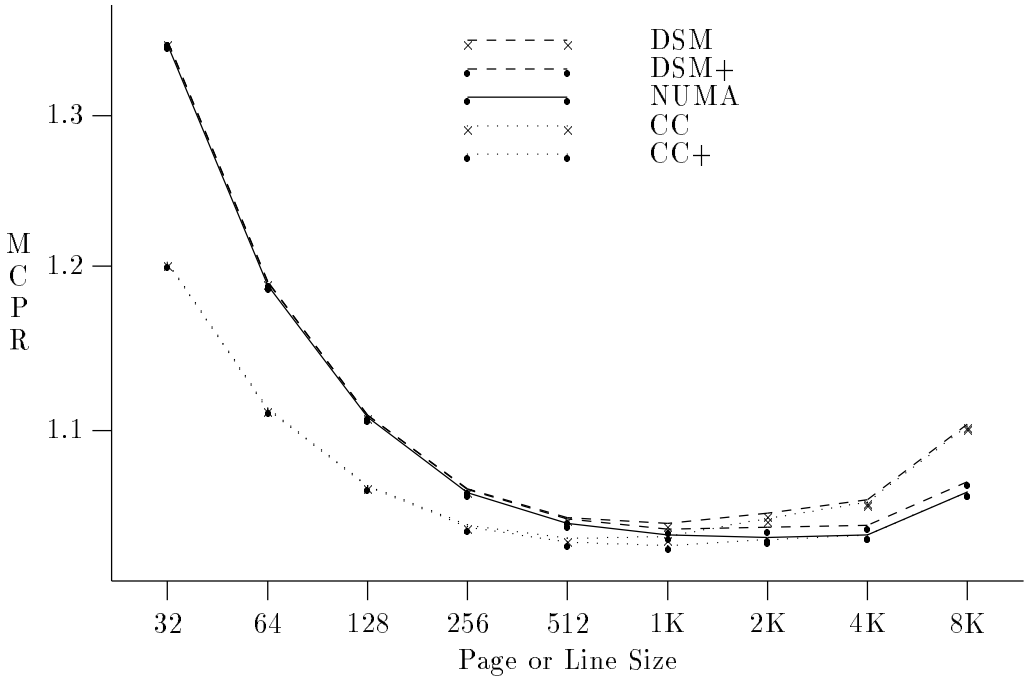

Figure 1: Mean cost per reference versus block size (log scales), for an SOR application.

2.1 The One-Word Block Definition

When asked to consider false sharing in the context of our memory cost model, several people initially suggested defining it to be the difference in cost between running the optimal off-line policy with a given block size and running it with one-word blocks. Indeed, when a trace is run with a single-word block size there is manifestly no false sharing. Furthermore, the optimal placement for a program with a single-word block size will have the property that as few words are transferred between processors as is possible while maintaining coherence. However, it does not result in the minimum *number* of transfers; most programs have at least some spatial locality of reference, and so would benefit from some grouping of transfers. In any reasonable set of machine models, the cost of moving a block must include both a per-byte bandwidth component and a per-message overhead component. Reducing the block size to one word minimizes the data transferred, but increases the number of operations and thus the overhead incurred. So, the naïve definition of false sharing could easily result in a negative amount of false sharing if the additional overhead generated outweighs the eliminated false-sharing induced coherence operations.

This effect can be seen in figure 1, which displays results for a successive over-relaxation (SOR) application. Performance is reported as mean cost per reference (MCPR), where a cost of 1 is defined to be the time to make a local cache hit. All performance figures in this paper include all memory references made by the program, both to private and explicitly shared memory. These results were obtained by running our off-line optimal policy over a 104 million-reference trace. The five curves represent five different sets of machine parameters. CC (cache-coherent) is a write-invalidate, sequentially consistent coherently cached shared memory multiprocessor; CC+ is a hypothetical cache-coherent machine that adds the ability to read or write data at a remote node without replicating the cache line. NUMA (non-uniform memory access) is meant to be similar to the Cray T3D, with

VM-based software coherence. DSM (distributed shared memory) is meant to be similar to VM-based software coherence on a machine like the Intel Touchstone Delta; DSM+ adds fault-driven software emulation of remote memory references. As noted above, the "optimal" policy doesn't actually optimize anything on the CC and DSM models, which lack remote reference capabilities. All five sets of parameters assume equivalent hardware technology. Further details appear in [4].

The total cost of memory references and coherence operations in the SOR program increases markedly as the block size is lowered toward one word. If all that happened as the block size was reduced was that false sharing was also reduced, then the cost would get smaller with the block size. However, exactly the opposite happens: the cost gets larger with a smaller block size. The sharing in this particular application is essentially migratory in nature: four kilobyte chunks are passed between processors, and performance suffers if this sharing happens by moving small pieces one at a time.

Most of the applications we studied suffer some degree of increased cost with small block sizes, due to the breakup of blocks that are not falsely shared. The effect is usually not as pronounced as in the SOR program, however, and is generally hidden by the reduction in false sharing; most applications have enough false sharing that reducing the block size is beneficial.

Because the one-word block definition of false sharing is unable to separate improvements in performance due to reductions in false sharing from degradations in performance due to increases in the number of operations needed, it fails the test for a proper definition; it does not adequately capture the intuitive notion of false sharing.

2.2 The Interval Definition

Imagine a coherence policy with perfect knowledge of the sharing behavior of the program (e.g. via perfect annotations provided by the programmer). Using this information, it would be possible to relax the implementation of sequential consistency: instead of requiring that only one copy exist at the time of a write, we could simply require that any time a read takes place, the reading processor sees the "freshest" data. Processors could have inconsistent copies of a (logically) single block, but would need to be able to re-merge these copies at some future time.

Define the effect of false sharing to be the difference in performance between the optimal policy using a traditional coherence constraint and the minimal cost achievable using the extended execution model with the new merge facility. This definition agrees with intuition, is mathematically precise, and describes a system that one could at least imagine implementing (given that the application writer or language tools provided good enough directives). It does not require heuristics or reasoning about the space of possible alternative programs and results in no fuzziness in the size of the measured effect, as do some of the other definitions presented later in this paper.

Unfortunately, this interval definition fails the practicality criterion: it is not known to be computationally tractable. Consider a string of references to a single block made by two different processors, as illustrated in Figure 2. Here, a notation like r_a^p means processor p read address a. A false sharing interval, then, is any interval that contains no pair of references w_a^p and r_a^q such that $p \neq q$, the write precedes the read, and the block is written and referenced by more than one processor during the interval. That is, a false sharing

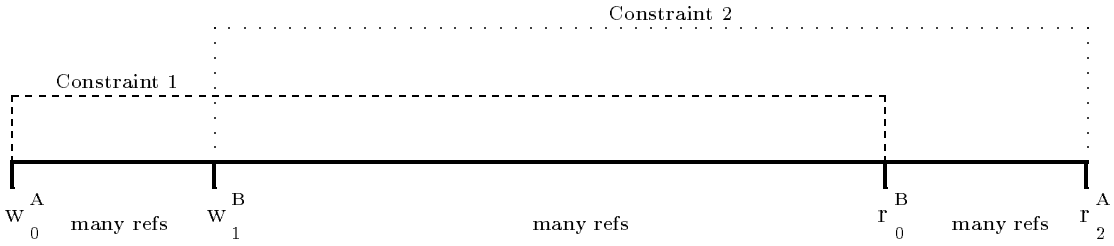

Figure 2: False sharing intervals.

interval is one in which the block is used by more than one processor, but in which no data communication takes place. A "maximal" false sharing interval is one that cannot be extended by one reference at either end, either because true data communication would be required, or because the end in question is at the beginning or end of the whole trace.

The situation shown in figure 2 has two potential "maximal" intervals: from just after the first write to just before the final read, or from some time in the past (determined by other constraints not shown) up to just before the first read. Neither of these intervals can be extended without violating some constraint, but yet they have a non-empty intersection and are not equal. The total number of possible "maximal" interval sets can be exponentially large in the number of references, and we know of no computationally efficient method to determine which interval set results in the lowest possible overall execution cost.

2.3 Heuristic Interval Selection

Given that there is no known way of optimally choosing false sharing intervals, it could still be possible to make a good guess as to which intervals to use. At the very least, even an arbitrarily selected interval set has the property that it provides a lower bound on the amount of false sharing present in an application. If the heuristic used is good, then the computed bound could be close to the real limit, and so might suffice to show that false sharing can be a large problem.

It is possible to have a (maximal) interval that has negative value: the cost of the coherence operations eliminated by the interval are smaller than the cost of merging the block at the end of the interval. Furthermore, the cost benefit of an interval depends not only on the references made during the interval, but also on the desired starting and ending locations of the single copy of the block at the beginning and end of the interval. These locations in turn depend on what other intervals are selected. So, even the simple heuristic of choosing some set of maximal intervals, simulating the trace and rejecting those intervals that have negative value can still result in the selection of intervals that increase cost rather than reduce it, because they add cost only after other intervals have been chosen. In fact, we tried just this experiment, and found that is was fairly common to have beneficial intervals become detrimental after intervals that were detrimental in the first place were removed.

The heuristic interval definition of false sharing fails the mathematical precision criterion; nevertheless, it is an approximation of the interval method that always errs in the direction of underestimating false sharing, and therefore provides a lower bound on false sharing. In practice, this lower bound is not very tight.

Munin's [7] software implementation of release consistency takes a practical approach to heuristic interval selection. Its designers observe that with the proper use of locks, any

references made by a processor to an object for which it holds a lock will be inside a false sharing interval. This interval is not necessarily maximal, but in practice it will often be sufficiently large to result in a significant reduction of false sharing. After a lock release, if another processor acquires a lock for an object in the same block (page), Munin uses a saved copy of the original version of the page to drive a diff-based merge operation. A similar approach was supported in hardware by machines from Myrias Research Corporation of Edmonton, Alberta, Canada.

2.4 Full Duration False Sharing

If an additional restriction is placed on the intervals used for defining false sharing, namely that they extend from the beginning of the trace to the end of the trace, then two helpful things happen. First, the interval selection problem goes away, because there can be only one maximal "full duration" interval. Second, the implementation question of how to deal with a program that has full-duration false sharing is much easier. All an implementor has to do, given that full duration false sharing is identified ahead of time, is to turn coherence off for the falsely shared regions of memory. Since by hypothesis no processor reads data written by another within a full-duration falsely shared block, sequential consistency will be maintained. There is no need for merge operations.

Along the same lines as full duration false sharing is the identification of words that are either only-read or are used by only one processor, but are located on a block that is written by other processors. If these words could be separated at compile time, then every access to them could be local.

We found that full duration falsely shared blocks are extremely rare. The result of exploiting full duration false sharing is sufficiently small as to be uninteresting in our application suite, and probably in most applications. While this definition is precise and complete, and describes something that can be called "false sharing," it fails to capture the real problem of false sharing. It is a valid definition of the wrong effect; the intuition criterion is not satisfied.

2.5 The Hand Tuning Method

Eggers and Jeremiassen [10] defined false sharing to be the cost of cache coherence operations that were initiated by a reference to a word that was not modified by any processor since it was last present at the referencing processor. This definition has the difficulty (which was not noted in [10]) that true sharing may be masked by such coherence operations. For example, in a system with two processors, A and B, a single block with three words, and a repeating reference pattern of the form $\mathrm{r}_2^B\ \mathrm{w}_2^B\ \mathrm{w}_1^B\ \mathrm{r}_0^A\ \mathrm{w}_0^A\ \mathrm{r}_1^A$,[1] Eggers and Jeremiassen's definition will identify all of the coherence operations as being due to false sharing, because they are all initiated by a processor reading a word that is never touched by the other processor. However, there is real data communication from processor B to processor A in word 1. In practice, the difference between false sharing as defined by Eggers and Jeremiassen and its "true" value may well be small, but it is difficult to determine if this is the case in any particular instance.

[1] Recall that a notation of the form r_x^p means processor p reads word x.

Eggers and Jeremiassen then proceed to measure false sharing in a second way: by hand-modifying their programs in order to reduce false sharing by applying a small set of transformations to the programs' source code. They ran and traced the modified programs, and again measured the number of cache coherence operations. They claimed that the difference in performance between the original and modified programs was the effect of false sharing. This latter definition is interesting in that it results in a practically achievable performance improvement. However, there is no guarantee that their transformations eliminated all of the false sharing in the program (or even that they did not reduce the amount of data communication for other reasons, such as reducing fragmentation in cache lines), and so is not mathematically precise.

They found reductions in overall bus utilization due to their false sharing removal transformations of up to about 25%, for cache lines no larger than 64 bytes.

3 The Cost Component Method

The cost component method is not a complete definition of false sharing. However, it appears to be a promising new direction from which a complete definition may be found. If the cost component method was completed, it would satisfy all three criteria and would constitute a proper definition.

A remote operation (either moving a block or making a direct remote memory reference) can be thought of as the sum of two costs: the cost of moving the data across the interconnect, and the cost of setting up the transfer. The first is known as the *data movement component* and the second as the *overhead component*. The data movement component depends only on the bandwidth of the interconnect and the number of words moved (but *not* on the number of separate transfers used to move the words). Overhead can depend on many factors. The cost component definition of false sharing uses the fact that changing the block size will affect the two cost components differently. First, we observe what happens to optimal performance when the block size is reduced, and then consider the relationship of these effects to the cost components.

In general, when the block size is reduced, the coherence operations performed by an optimal policy will change. These changes will be due to one of three effects:

1. If data that are used together are separated, more coherence operations will be necessary to move them.
2. If falsely shared data are separated into pieces that are no longer falsely shared, coherence operations will be eliminated.
3. If a block is split into two pieces and only one of those pieces is used, the cost of moving the other piece will be saved.

Or, more concisely, overhead will increase for moving large blocks of data that should be grouped together; false sharing will be reduced; and fragmentation will be reduced. The combination of these three effects results in the net change in cost between the initial and smaller block size systems. Define the amount of false sharing at block size s to be the difference in the value of the false sharing component between a run at block size s and

a run with a single word block size. (One word block size machines thus have no false sharing).

Breaking up data that are used together ("useful groupings") results in increased cost because more operations are required to accomplish the same task. All of this additional cost will show up as additional overhead; the number of bytes transferred through the interconnection network will not change. Reduction of false sharing reduces the total number of operations necessary, thus reducing both the number of bytes transferred and the amount of overhead incurred. Reducing fragmentation does not affect the number of operations, and so produces no change in the overhead, but reduces the volume of data transferred.

If we define S to be the false sharing component and F the fragmentation component of the difference in cost between runs with regular and single-word blocks, we can show [6] that the grouping components cancel out, and

$$S = (o + bs)M_s - (\frac{o}{s} + b)M_1 - (1 + \frac{o}{bs})F \quad (1)$$

where o is per-message overhead, b is per-byte overhead, s is block size, and M_s and M_1 are the number of block moves performed by an optimal policy with block sizes of s and 1, respectively.

Unfortunately, there does not seem to be any obvious way to measure the amount of fragmentation in a program independent of false sharing. The best we can do, since fragmentation can never increase with smaller blocks, is to set F to zero in Equation 1, thereby obtaining an upper bound on false sharing:

$$S \leq (o + bs)M_s - (\frac{o}{s} + b)M_1 \quad (2)$$

The cost component definition measures the total amount of performance improvement that could conceivably be obtained by eliminating false sharing, assuming that it were possible to do so without increasing overhead by using smaller block sizes, or merge operations. The only plausible way of doing such a thing is by directive from the application. Getting the kinds of savings shown by this method would require either very careful application tuning, or a very good compiler, library and/or runtime tools to assist the coherence policy in making its decisions.

4 Estimating False Sharing

None of the definitions discussed in sections 2 and 3 provides a way of measuring false sharing. Equation 1 from the cost component method could be used to show false sharing as a function of F, but the possible range of false sharing thus demonstrated is very large: for applications we have studied, the contribution of false sharing to the total cost of shared memory could range from nothing to over 90% of total cost.

To get a hint as to where in this range false sharing really lies, we can consider how much of the total memory (reference and block move) cost is due to per-message overhead, and how much is due to per-byte data transfer costs. Figures 3, 4, 5 and 6 provide this breakdown for four of our applications, again using trace-driven simulation of an optimal placement policy, with two different sets of machine parameters. Recall that MCPR is the mean cost of a memory reference expressed in terms of a local cache hit, and that all graphs

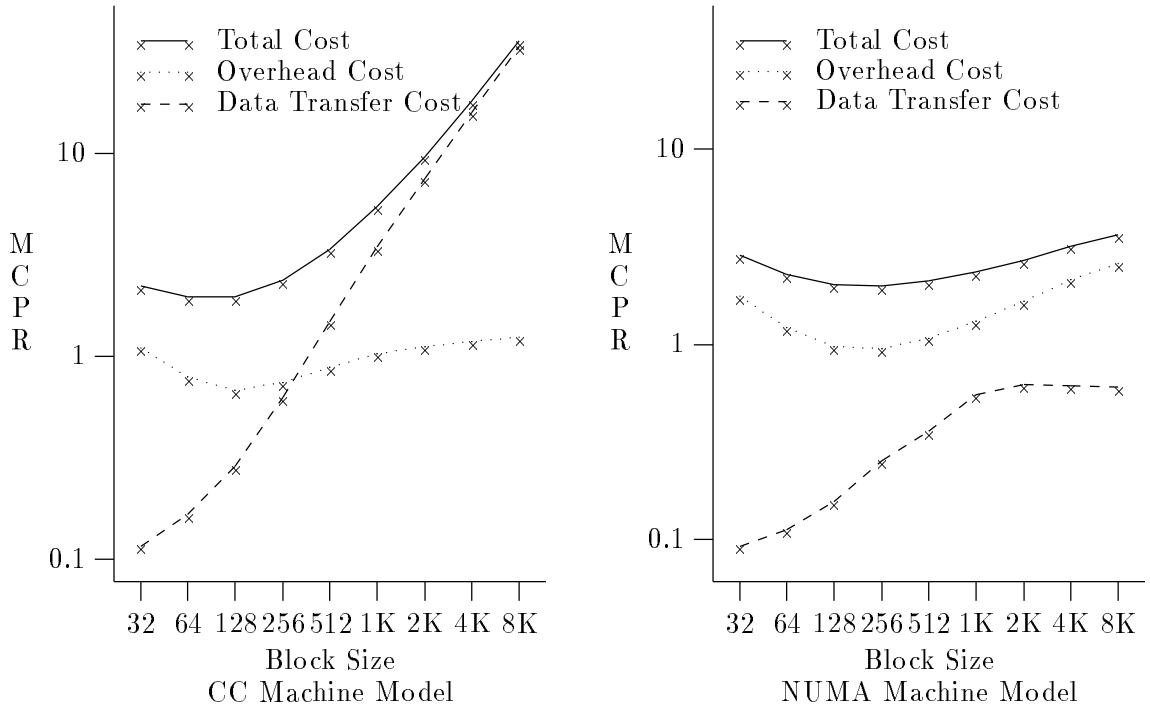

Figure 3: Data transfer and overhead components (log scales) of mean cost per reference for a scene-rendering application.

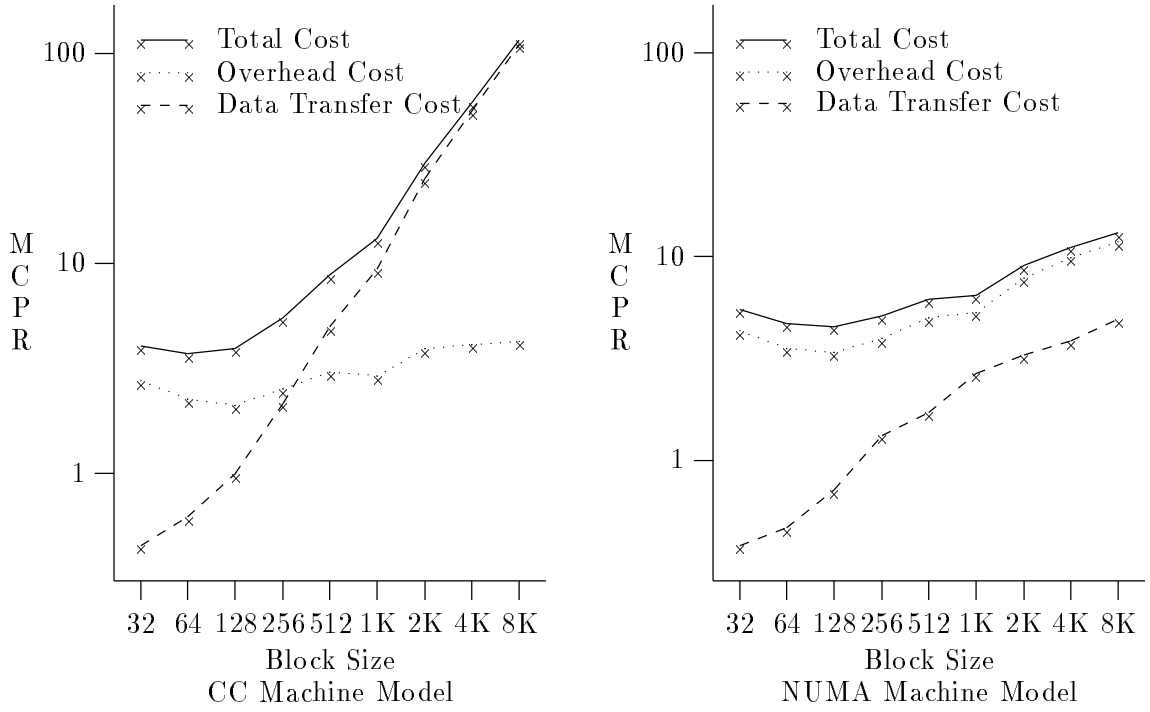

Figure 4: Data transfer and overhead components (log scales) of mean cost per reference for parallel quicksort.

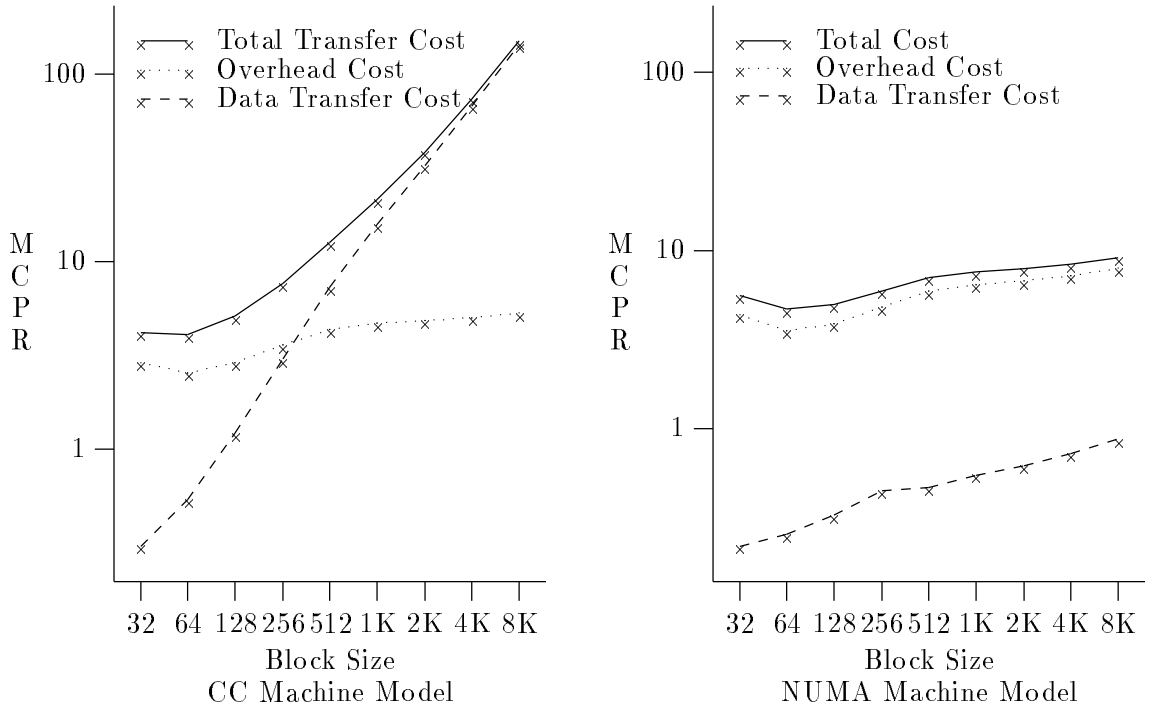

Figure 5: Data transfer and overhead components (log scales) of mean cost per reference for Cholesky factorization.

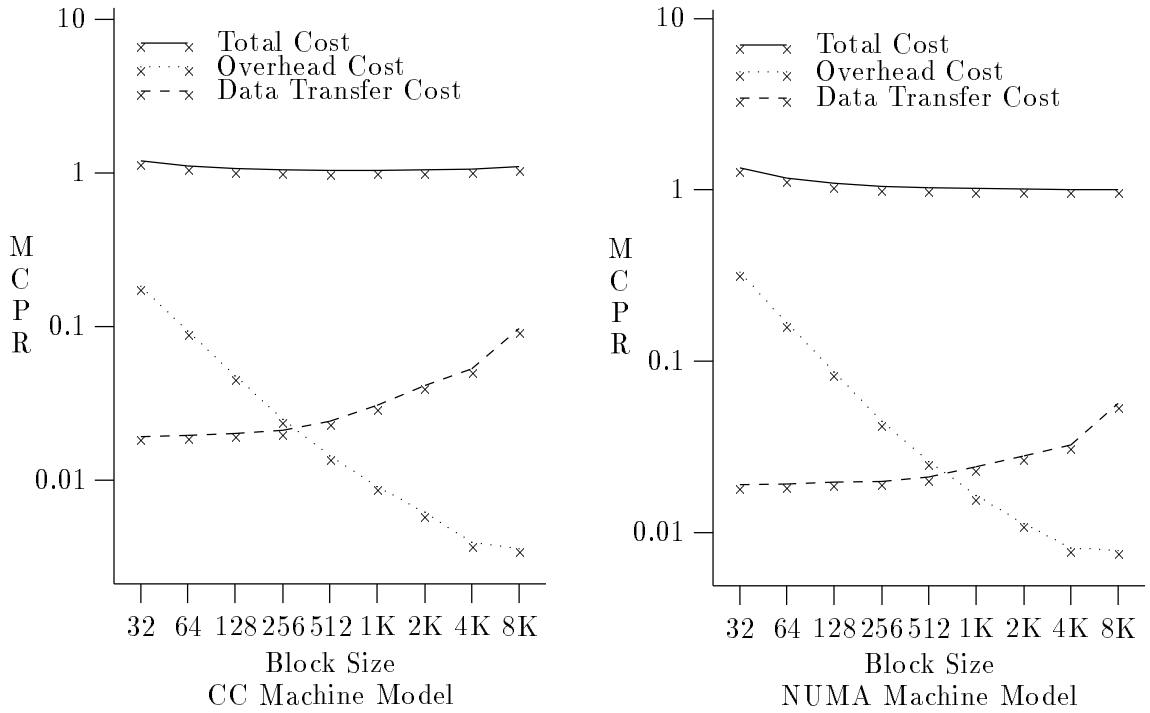

Figure 6: Data transfer and overhead components (log scales) of mean cost per reference for an SOR application.

in this paper include all memory references: both those made to shared and explicitly private memory. The applications are a scene-rendering program [11], a Presto [1] quicksort program, the Cholesky factorization program from the SPLASH suite [17], and our successive over-relaxation program. The machine parameters model a sequentially consistent cache-coherent (CC) multiprocessor, and a VM-based software coherence system running on a non-uniform memory access (NUMA) machine resembling the Cray T3D.

The "useful grouping" cost component can only result in increased cost with reduced block sizes. Fragmentation does not affect the overhead component at all. Therefore, any reduction in overhead with reduced block size must be due to a reduction in false sharing. Furthermore, that reduction in overhead must be accompanied by some reduction in data transfer as well, because the reduction in false sharing results in fewer total transfers needed, and a transfer incurs both overhead and data transfer costs.

When the block size is halved, we can show that fragmentation can at most be reduced by a factor of two. By induction, when block size changes from s to $\frac{s}{2^n}$, fragmentation can reduce the data transfer cost component by at most a factor of 2^n. (The result generalizes to denominators that are not powers of two.) Many of our applications approach or even exceed this limit over a wide range of block sizes. For example, the data transfer cost component for the quicksort program with 8K blocks on the CC model is 1.2 billion cost units. At a 1K block size the data transfer cost is 138 million, for a ratio of 8.7, which is greater than the factor of 8 that can be explained by fragmentation alone. At a 128 byte block size the data transfer cost is 28 million cost units, or 43 times less than that at 8K; while this could be explainable entirely by fragmentation, doing so would mean that nearly all of the memory in the 8K blocks was unused. This is very unlikely. False sharing is almost certainly responsible for most of the difference in performance (which is greater than an order of magnitude).

On the other hand, the SOR application has very little false sharing. Its data transfer cost component is never greater than 10% of the total cost of running the application.[2] The ratio of the data transfer cost at an 8K block size to that at a 32 byte block size is 5 to 1, which is much less than the factor of 256 that could be accounted for by fragmentation. The overhead component steadily increases in its contribution to cost as the block size is reduced, in stark contrast to the other applications discussed.

Most of the applications in our suite display performance like that of the quicksort, scene rendering, and Cholesky programs, rather than the SOR program. This indicates that false sharing is probably the major reason for the poor performance of these applications on large block size machines. If false sharing were somehow reduced, our results suggest that machines with page-size blocks would perform comparably to those whose blocks are the size of a typical cache line.

5 Conclusion

The impact of false sharing on parallel program performance depends on many factors, including block size, data layout, program access patterns, and the cost of coherence operations. Quantifying this impact has proven surprisingly difficult. Program or machine

[2] The total cost as shown in these graphs is the sum of not only the data transfer and overhead components, but also the cost of making local memory references, which contributes 1 to the mean cost per reference.

changes that serve to reduce false sharing also tend to affect other components of memory system cost, including the amortized overhead of large bulk transfers and the data transfer costs of internal fragmentation. We know of no effective way to separate false sharing from these other effects. Moreover, even when program or machine changes serve to improve performance, we know of no way to determine definitively whether the improvements are large or small relative to what is theoretically achievable.

From a purely conceptual point of view, the most precise and reasonable characterization of false sharing would appear to be the interval definition, described in section 2.2. In essence, this definition says that the cost of false sharing is the difference in performance between a policy that makes optimal placement decisions, but that enforces consistency on a whole-block basis, and one that enforces consistency only in the event of genuine word-level data dependences. Unfortunately, the temporal overlapping of dependences leads to a combinatorial explosion of possible placements, suggesting that measuring false sharing under this definition is probably NP-hard.

Several less conceptually satisfying definitions of false sharing lend themselves to actual measurement, including definitions based on heuristic interval selection, full duration false sharing, hand tuning, and the cost component method. Combining analysis from the cost component method with the results of trace-driven simulation, we find that the improvements in performance that result from smaller blocks approach or even exceed the maximum possible effect of everything other than false sharing. When combined with some knowledge of application semantics, these results suggest that the elimination of false sharing could result in order-of-magnitude improvements in performance for many programs.

Relaxed models of memory consistency (as in DASH [14] or Munin [7]; see [15] for a survey) constitute one promising approach to reducing the impact of false sharing. In essence, relaxed consistency models suffer delays due to false sharing only at synchronization points. Other means of reducing false sharing include on-line adaptation of the block size [9], hand tuning [10], and smart compilers. Each of these approaches has its drawbacks, but the potential gains appear to be large enough to warrant substantial investments in hardware or in software.

6 Availability

Compressed postscript for this paper and other systems papers from the University of Rochester Computer Science Department may be obtained by anonymous ftp from `pub/systems_papers` on `cs.rochester.edu`. Printed versions of technical reports from URCS may be obtained for a fee by contacting `tr@cs.rochester.edu`, or through physical mail from Technical Reports Librarian/Department of Computer Science/University of Rochester/ Rochester, NY 14627-0226. Under very special circumstances, the traces used in this paper may be made available. However, making copies of these traces requires substantial personal effort on the part of the author, and so will not be undertaken lightly or often. If you feel that you have a real need for the traces, write to Bill Bolosky.

References

[1] B. N. Bershad, E. D. Lazowska, H. M. Levy, and D. B. Wagner. An Open Environment for Building Parallel Programming Systems. In *Proceedings of the First ACM*

Conference on Parallel Programming: Experience with Applications, Languages and Systems, pages 1-9, New Haven, CT, 19-21 July 1988. In *ACM SIGPLAN Notices 23*:9.

[2] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott. Simple But Effective Techniques for NUMA Memory Management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19-31, Litchfield Park, AZ, 3-6 December 1989. In *ACM SIGOPS Operating Systems Review 23*:5.

[3] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212-221, Santa Clara, CA, 8-11 April 1991. In *ACM SIGARCH Computer Architecture News 19*:2, *ACM SIGOPS Operating Systems Review 25* (special issue), and *ACM SIGPLAN Notices 26*:4.

[4] W. J. Bolosky and M. L. Scott. A Trace-Based Comparison of Shared Memory Multiprocessor Architectures. TR 432, Computer Science Department, University of Rochester, July 1992.

[5] W. J. Bolosky and M. L. Scott. Evaluation of Multiprocessor Memory Systems Using Off-Line Optimal Behavior. *Journal of Parallel and Distributed Computing*, 15(4):382-398, August 1992.

[6] W. J. Bolosky. Software Coherence in Multiprocessor Memory Systems. Ph. D. Thesis, TR 456, Computer Science Department, University of Rochester, May 1993.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152-164, Pacific Grove, CA, 14-16 October 1991. In *ACM SIGOPS Operating Systems Review 25*:5.

[8] A. L. Cox and R. J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 32-44, Litchfield Park, AZ, 3-6 December 1989. In *ACM SIGOPS Operating Systems Review 23*:5.

[9] C. Dubnicki and T. J. LeBlanc. Adjustable Block Size Coherent Caches. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 170-180, May 1992.

[10] S. J. Eggers and T. E. Jeremiassen. Eliminating False Sharing. *Proceedings of the 1991 International Conference on Parallel Processing*, I, Architecture:377-381, August 1991.

[11] A. Garcia. Efficient Rendering of Synthetic Images. Ph. D. thesis, MIT, February 1988.

[12] A. Garcia, D. Foster, and R. Freitas. The Advanced Computing Environment Multiprocessor Workstation. Technical Report RC-14419, IBM T. J. Watson Research Center, March 1989.

[13] R. P. LaRowe, Jr. and C. S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319-363, November 1991.

[14] D. Lenoski, J. Laudon, L. Stevens, T. Joe, D. Nakahira, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, May 1992.

[15] D. Mosberger. Memory Consistency Models. *ACM SIGOPS Operating Systems Review*, 27(1):18-26, January 1993. Relevant correspondence appears in Volume 27, Number 3; revised version available Technical Report 92/11, Department of Computer Science, University of Arizona, 1993.

[16] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52-60, August 1991.

[17] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *ACM SIGARCH Computer Architecture News*, 20(1):5-44, March 1992.

[18] J. Torrellas, M. S. Lam, and J. L. Hennessy. Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates. *Proceedings of the 1990 International Conference on Parallel Processing*, II – software:266-270, August 1990.