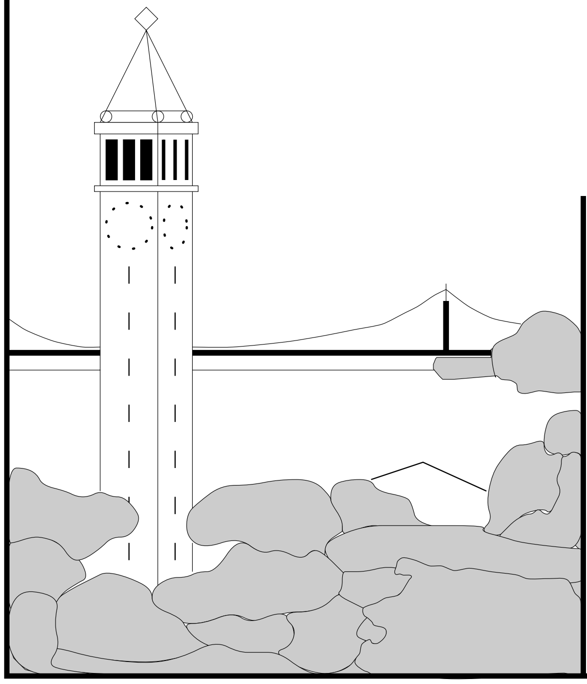
Analysis of Shared Memory Misses and Reference Patterns

Jeffrey B. Rothman and Alan Jay Smith



Report No. UCB/CSD-99-1064

September 1999

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Analysis of Shared Memory Misses and Reference Patterns[†]

Jeffrey B. Rothman and Alan Jay Smith
Computer Science Division
University of California
Berkeley, CA 94720

September 23, 1999

Abstract

Shared bus computer systems permit the relatively simple and efficient implementation of cache consistency algorithms, but the shared bus is a bottleneck which limits performance. False sharing can be an important source of unnecessary traffic for invalidation-based protocols, elimination of which can provide significant performance improvements. For many multiprocessor workloads, however, most misses are true sharing and cold start misses. Regardless of the cause of cache misses, the largest fraction of bus traffic are words transferred between caches without being accessed, which we refer to as *dead sharing*.

We establish here new methods for characterizing cache block reference patterns, and we measure how these patterns change with variation in workload and block size. Our results show that 42 percent of 64-byte cache blocks are invalidated before more than one word has been read from the block and that 58 percent of blocks that have been modified only have a single word modified before an invalidation to the block occurs. Approximately 50 percent of blocks written and subsequently read by other caches shown no use of the newly written information before the block is again invalidated.

In addition to our general analysis of reference patterns, we also present a detailed analysis of false sharing and dead sharing in each shared memory multiprocessor program studied. We find that the worst 10 blocks from each our traces contribute almost 50 percent of the false sharing misses and almost 20 percent of the true sharing misses (on average). A relatively simple restructuring of four of our workloads based on analysis of these 10 worst blocks leads to a 21 percent reduction in overall misses and a 15 percent reduction in execution time. Permitting the block size to vary (as could be accomplished with a sector cache) shows that bus traffic can be reduced by 88 percent (for 64-byte blocks) while also decreasing the miss ratio by 35 percent.

[†]The work presented here has been supported in part by the State of California under the MICRO program, Sun Microsystems, Toshiba Corporation, Fujitsu Microelectronics, Cirrus Corporation, Microsoft Corporation, Quantum Corporation, and Sony USA Research Laboratories.

1 Introduction

Shared memory multiprocessor systems are becoming increasingly popular. The limit to the number of processors that can be placed on the same memory bus is due to the bus traffic demands of the processors. Here we present a new examination of the interference patterns of references to words within shared blocks, with the purpose of aiding both software developers in data layout and hardware designers in the development of new protocols that perform coherence (cache consistency) on a subblock basis. Our purpose is to examine the causes of bad behavior in parallel programs, aiming to reduce bus traffic and miss ratios. This study uses relatively large traces of twelve parallel workloads to provide our results. We measure the sharing behavior of words within shared blocks to determine the extent that false sharing occurs. We also look at the related phenomenon of *dead sharing*, which is determined by measuring the words within a block that are not utilized while in the cache; as will be shown, these words consume the largest proportion of bus traffic.

The remainder of this paper is organized as follows: the next section describes our motivation for undertaking this study. Section 2 provides an overview of related work in the area of characterization of sharing patterns of parallel programs. Section 3 discusses our methodology for creating and evaluating the parallel memory traces and describes some of the metrics we use to measure the underlying behavior that causes shared memory traffic problems. In Section 4 we present our results and discuss our observations. Section 5 summarizes our conclusions.

1.1 Definitions

Table 1 provides the definitions of the terms we use throughout this paper.

Definitions Used in This Paper	
Shared Memory	The portion of the memory space that is visible to all processors.
Block	A group of sequential memory locations that are fetched and evicted from caches together, aligned so that the address of first byte of the block has $log_2(Block\ Size)$ zeros as the lowest order bits.
Shared Block	A shared block generally is a block from shared memory; more specifically, a block that is referenced by more than one processor during the execution of a program.
Private Block	A block that is accessed only by one processor for the duration of the simulation.
Shared Access	A memory reference to a block that at some point during the simulation is considered to be a shared block. Note: an access that is classified as shared in one simulation may not be shared in a simulation with a smaller block size.
Actively Shared Memory	The set of blocks that are accessed by multiple processors for a given set of simulation parameters.
Global Unshared Access	Access to a portion of memory that is declared to be shared, but is used exclusively by a single processor.
Private Access	A memory reference to a private block.
False Sharing	False sharing occurs when two or more data items that are unrelated happen to be placed in the same block, causing an unnecessary increase in bus traffic to maintain coherence. In this paper we define false sharing as a reference to a word in a block, finding it to be in a different state in a particular block under block coherence and transfer size than under word granularity coherence/transfer size. Thus it can have a good side-effects (such as in the prefetching effect of large blocks), or bad side-effects, such as extra misses and coherency operations.
Dead Sharing	The portions of the block that are not referenced while in the cache, resulting in wasted bus traffic.
Local Read	Read by the processor that has most recently written the block.
Local Write	Write by the processor that has most recently written the block.
Remote Read	Read by any processor except the one that has most recently written the block. This also includes reads by processors with their own copies of the block.
Invalidation Interval	The string of references to a block by all processors between coherence induced invalidations.
Span	Distance between the furthest apart words in a block that are referenced during an invalidation interval.
Processor-spatial	The footprint of accesses by a processor during an invalidation interval.
Cache Hit	The data requested is found in the cache and the processor can proceed without causing a coherence operation.
Fetch Miss (**fmiss**)	A reference to the cache which does not find the requested data, requiring a fetch from main memory. or another processor's cache.
Invalidation Miss (**imiss**)	A write reference to a block (or word) held in the cache in the **shared** state, requiring the processor to stall while invalidating copies of the data in other caches (under sequential consistency).
Miss	Both fetch and invalidation misses.
reference run	the stream of uninterrupted references by one processor to a block.
read run	a reference run consisting purely of reads
read/write run	a reference run consisting of reads and writes
write run	the stream of writes in the read/write run

Table 1: Definitions used in this paper.

1.2 Motivation

In the process of determining what sort of memory accesses caused the most traffic in our workloads, we examined the source code to identify which data structures were responsible for the most unfortunate sharing patterns (detailed in Section 4.4 and Appendix B). Some of the programmer specific details that have come out of this research are described in more detail in Section 4. During our analysis we found several recurring types of data structures which seem to lead to bad data access patterns.

When the programmer is not sufficiently careful in his or her data layout, it is necessary for some combination of the compiler and hardware to try to minimize coherence induced traffic. This paper investigates the sources of traffic caused by inadvertently poor data organization and provides suggestions for solving these problems.

Our goal in this research is to uncover the effects of poor programming style and to provide information about how these problems can be corrected. Additionally, we want to show how all types of sharing impact the performance of these workloads, and to demonstrate the degree to which block size affects spatial locality.

2 Background

Previous research on multiprocessor reference patterns has primarily focused on evaluating various cache coherence protocols for suitability, primarily contrasting invalidation- and update-based algorithms. The initial papers in this area ignored block size altogether, exclusively using 4-byte blocks to examine primarily the patterns of writing references [EK88, AG88]. The reference patterns were categorized by the length of the *reference run*. A reference run can be further refined into *read runs*, *read/write runs*, and the *write run* (Table 1). The write-run lengths varied widely between applications, leading to inconclusive results whether update- or invalidate-based protocols give superior performance.

Research concerning *reference runs* for different block sizes found that for scalar (non-vector) workloads, the lengths of the various reference runs did not increase with block size, although vector workloads showed improved processor locality with larger block sizes [GS94]. The poor locality in scalar workloads was attributed to fine-grain sharing of data among the processors. A study of the effect of block size on data structures concluded that the excessive invalidations are caused by a mismatch between data objects and block size [GW92].

3 Methodology

Our work is based on trace-driven simulation (TDS). Initially we used execution-driven simulation for our research, but we changed to TDS for two reasons: (1) the locations of data objects varied as parameters such as block size changed, making detailed analysis very complicated; and (2) our EDS tools are dependent on generally obsolete DEC 5000 machines; using traces allows for much more rapid generation of results on faster PCs and workstations. We used our EDS tool Cerberus [RS99b] to generate traces for the simulation system. The trace generation system simulates synchronization objects (locks and barriers) at runtime, which aids in reducing trace length (by eliminating spin-waiting loops in the trace file) and allows more accurate synchronization behavior while producing traces.

3.1 Simulation

Our simulations use infinite size fully-associative caches to eliminate capacity and conflict misses, to focus on the effect of coherency induced misses and traffic. The remaining misses fall into three categories: cold start/private, cold start/shared, and coherency caused misses. A reference to a private block causes a single cold start miss, which is the only miss for the rest of the simulation; shared blocks can have up to 16 cold start misses (with 16 processors reading them in for the first time), subsequent shared misses are either caused by true or false sharing. The cache simulators tested parallel workloads with 16 processors and block sizes ranging from 4 to 512 bytes.

3.2 Workload Characterization

Twelve parallel programs were examined to provide the results for this paper. Ten of the programs come from the SPLASH suites from Stanford University, which have been available to the research community as a de facto benchmark for comparing parallel program execution. These programs have all been used in a number of studies analyzing parallel code performance and are characterized and described in more detail in [SWG92, WOT$^+$95]. The other two programs used in this study (**topopt** and **pverify**) were created by the CAD group at U.C. Berkeley and have been used for measurements at Berkeley and the University of Washington [EK89a, EK89b, EJ91, AB95]. Detailed descriptions of the workloads can be found in Appendix A. All workloads were traced on a MIPS R3000 based workstation using the **Cerberus** multiprocessor simulator [RS99b]. Each workload is traced from beginning to end to capture the entire behavior of the program.

Program Characteristics									
Programs	References (Millions)		Shared Data	Private Data	Shared			Private	
					Fraction of Data References				
	Inst	Data	(KBytes)		Reads	Writes	Locks	Reads	Writes
barnes	114.23	42.31	33.02	34.64	0.159	0.005	0.002	0.456	0.379
cholesky	90.67	34.32	970.02	783.38	0.496	0.075	0.013	0.287	0.130
fmm	288.30	166.82	380.41	460.14	0.136	0.006	0.000	0.347	0.511
locus	805.62	164.45	1405.70	1151.79	0.563	0.020	0.001	0.255	0.162
mp3d	174.88	60.82	701.91	181.53	0.318	0.223	0.001	0.302	0.157
ocean	234.12	92.38	140.16	984.45	0.264	0.031	0.014	0.560	0.132
pthor	275.87	97.77	1233.09	1026.75	0.384	0.047	0.049	0.350	0.176
pverify	181.29	55.24	23.08	149.67	0.473	0.015	0.008	0.320	0.186
raytrace	471.13	196.96	667.30	2144.09	0.318	0.003	0.002	0.429	0.248
topopt	655.75	141.60	19.22	38.76	0.812	0.087	0.000	0.085	0.016
volrend	351.62	79.92	395.61	2340.98	0.477	0.007	0.003	0.287	0.226
water	366.23	127.67	44.51	102.37	0.179	0.016	0.002	0.577	0.227
Total					Average				
Overall	4009.7	1260.3	6014	9399	0.381	0.044	0.008	0.354	0.213

Table 2: Reference characteristics of workloads for 16-processor simulation, using 4-byte blocks.

Table 2 shows the reference characteristics for a 16 processor 4-byte block simulation of the various workloads on our SMP simulator, which runs on uniprocessor workstations. The number of shared references in Table 2 were measured using 4-byte blocks, which captures the number of truly shared words. Global unshared references and private references (Table 1) are lumped together under the *private* heading. The fraction of shared accesses has quite a large variation; it ranges from 0.16 in **barnes** to 0.90 in **topopt**, with an average of 0.43 for all workloads. However, as the block size increases, memory locations and references which are classified private in Table 2 can become shared, so it is necessary to trace all references which are to shared memory. We also tracked all references to private memory to understand its contribution to total memory traffic. As the cache simulators were designed to make the common transactions very quick through the use of hashing, tracking the references to private memory is generally not a major contributor to simulator execution time.

3.3 Metrics

The traditional reference stream interval used for measuring sharing behavior is the *reference run* [EK88, AG88, GS94]. Along with related measures such as the *read run*, *read/write run*, and the *write run* (Table 1), it can provide some idea of the residency time of a block in one processor's cache (processor locality) and the appropriateness of coherence protocol (invalidate vs. update).

A major problem with the reference run as a metric is the lack of information concerning how the processors share data within blocks. It is possible to get a crude idea of contention by examining the length of these different types of reference runs, but they provide no indication about the type or granularity of sharing within a block. We establish a new reference stream interval called an *invalidation interval*, which is the string of references to a block by all processors between coherence induced invalidations (Figure 1). This allows a longer term and much more detailed study of dynamic sharing behavior within a block.

The metrics we describe here are all concerned with the *processor-spatial* properties of multiprocessor programs. By this term we mean the number of unique words within a block that are accessed while it is valid in a cache. This concept also encompasses measuring the fraction of the block which contains these words, which we refer to as the *span*. By using these types of metrics, it is possible to get an idea of the typical range of block sizes that make sense to use with multiprocessor caches.

The results presented later in this paper demonstrate that a variable block size (or use of subblocks) can significantly outperform any fixed block size for all the workloads examined, reducing traffic by 88 percent while decreasing the miss ratio by 35 percent (on average) for 64-byte blocks.

We can think of an invalidation interval as having two phases: (1) a write phase and (2) a read phase. The write phase consists of local reads and writes, which cause no bus activity after the first write (assuming write-allocate). The read phase (if there is one) begins with the first remote read, and consists of local and remote reads. We use the statistics of the references to individual words during invalidation interval to evaluate processor-spatial lo-

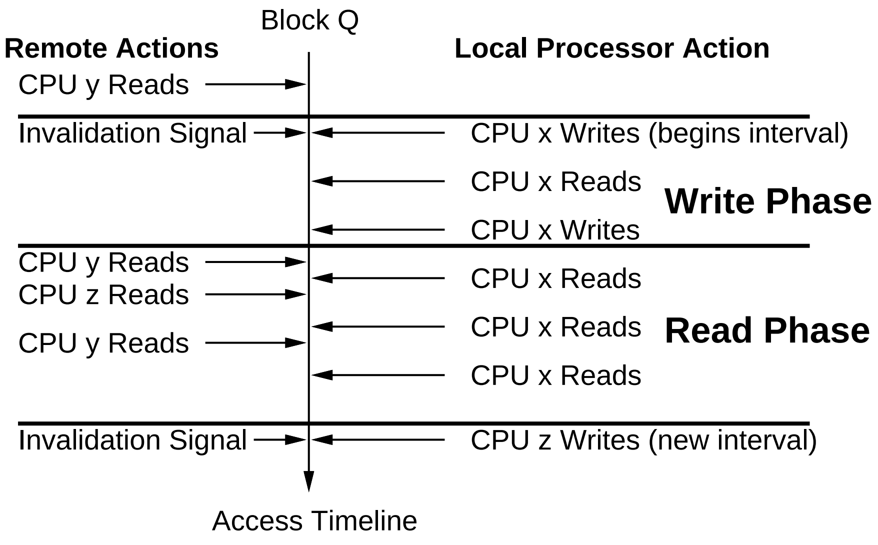


Figure 1: An invalidation interval is a string of references to a block, lasting from the first invalidation to a block until the next invalidation. During the interval other processors may read the block, but not write it.

cality, which we will show is generally rather poor, i.e., large block transfers for shared data are demonstrated to be wasteful.

Our goal is to provide the tools to aid in improving spatial locality in shared memory systems, or at least provide insight into the lack of spatial locality. It has been demonstrated that shared data in multiprocessor workloads have worse locality of reference than unshared data [EK89b], but increasing the block and cache sizes have not always provided a solution. It is necessary to understand what kind of misses are causing poor performance and examine the data structures/objects that produce such problems.

Implicitly our study assumes a write-invalidate protocol as backdrop against which our analysis is done. Invalidation-based protocols logically offer a better solution to bus-based systems, due to the necessity of reducing traffic over the shared bus to memory. A pure update protocol (update on each write to a shared block) uses an estimated 2-25 times the traffic of write-invalidate protocols for coherence related operations [Lil93], and the amount of network traffic increases with cache size. This is caused by the requirement to update on each write to shared cache blocks, regardless of the age (staleness) of the block in the cache; this problem worsens as the number of blocks in the cache increases, generating the most bus traffic for infinite caches. There are some adaptive protocols that allow switching between update and invalidate for each block depending on the access pattern for the block; but of the non-adaptive coherence protocols, invalidation-based protocols typically outperform update-based protocols [GS96]. Additionally, write-invalidate protocols are the most popular class of protocols that are actually implemented in real systems [Ste90, HP96], which makes them a more attractive target for performance improvement. When a program is properly (re)structured to reduce the movement of blocks between processors, write-invalidate based protocols provide better overall performance.

4 Results

This study examines SMP (symmetric multiprocessor) memory access behavior on three levels, which successively refine the granularity of the inspection to smaller features. The first and coarsest level of analysis looks at the aggregate behavior of all of the memory references. Misses are broken down into true and false sharing fetch and invalidation misses, and the bus traffic due to dead sharing is shown.

The second level of memory reference observation looks at the spatial reference pattern to shared blocks. This consists of examining the unique (distinct) words within a block that are referenced from the time it is read into the cache until the time it is invalidated. In addition, we look at the footprint of those words within the block, which we refer to as the **span**. The span provides a means of determining the spatial locality of a set of block references.

To develop a hardware protocol or software restructuring method to reduce bus traffic from coherency overhead, it is necessary to understand the patterns of sharing that occur and the competition of processors for words within shared blocks. At our finest grain level of examination, the words from the 10 worst (judged by misses) 64-byte blocks from each workload are characterized by the reference pattern for each individual word. This provides insight into the types of data objects which cause much of the traffic problems when they are placed in close proximity and provides hints into hardware and software solutions that can be used to eliminate or ameliorate much of the traffic/miss problem.

4.1 Gross Characterization of Misses

Most cache coherence protocols associate a set of states with each block in each cache, which are a subset of the MOESI family of protocols [SS86]. Each state consists of binary values for each of three attributes: validity (contains the most recent cached value of a block), exclusivity (only copy of a block), and ownership (the block in the cache possibly is inconsistent with main memory, but is the "correct" copy). The MOESI states consist of **M** (exclusive and owned), **O** (shared and owned), **E** (exclusive and unowned), **S** (shared and unowned), and **I** (invalid). A shared block is one which can be present in multiple caches; an owned block is inconsistent with memory, requiring a write-back by the "owner" processor at some point. All but the **I** state have the validity attribute associated with them. Almost any protocol (update, invalidate, or hybrid protocols) can be defined by a (sub)set of these states, plus the local and remote operations to the blocks that cause transitions between the various states.

Using a MESI (no **O** state) write-invalidation-based protocol with an infinite cache, we simulated a number of parallel programs maintaining two levels of granularity of coherence for blocks: for individual words and for the whole block, tracked in parallel during program execution. Once a block has been referenced, a valid copy of it exists in at least one cache for the rest of the simulation.

4.1.1 Types of Sharing Misses

False sharing has been studied by a number of researchers to measure the impact [TLH94, DSR$^+$93, BS93], as a concern for protocol developers [KB95, Lil93, Dah95], and as a target for data restructuring [EJ91, TLH90, JE95, HL90]. As measured in [TLH94], false sharing misses generally have a smaller impact on the miss ratio and bus traffic than true sharing misses. Our results show that false sharing misses become the largest source of misses for our workloads (on average) with blocks as small as 16 bytes.

False sharing occurs when two or more processors share a cache block, but access disjoint words or portions of the block. A classic example consists of two processors writing to their own distinct words within a block (Figure 2). The words are not shared, but since coherency

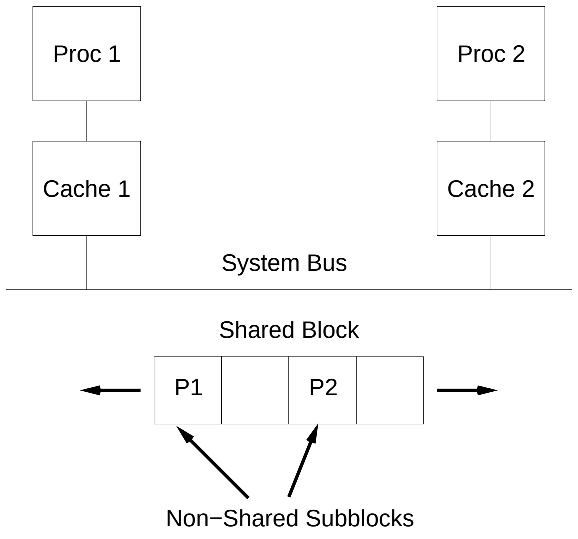


Figure 2: Simple schematic of false sharing. Two processors access disjoint words within a block, causing it to transfer back-and-forth between the caches (ping-ponging).

is enforced with block size granularity, the whole block is transferred back and forth between the processors' caches (in a process referred to as *ping-ponging*). In this situation, false sharing results in unnecessary data and coherence traffic to maintain cache consistency; it is not important that the data has become inconsistent, since none of the data is actually shared, just the block that contains them. The data does not need to be updated in the cache of the other processor (since it is never used); only the main memory needs to be properly updated when the block is written back on eviction of the block from the cache. Note that if the blocks, but not the words, have become inconsistent, when a modified block is copied back to main memory, only the modified words should be updated in main memory; otherwise main memory will not be properly updated.

Misses due to false sharing are easy to determine in a block with a reference behavior such as in Figure 2 when each processor accesses its own disjoint set of distinct words in a block. In that situation all misses except the initial compulsory misses are false sharing misses. In most cases, a determination of the type of sharing that is occurring requires reference by reference analysis of block and word states, as the access patterns of each processor to a block vary over time.

To measure false sharing in less obvious situations, it is necessary to examine the how the state of a word is affected by it being included with other words into a block granularity coherence unit. In our simulations, we perform this task by maintaining coherence state for each block and simultaneously for each word in the block. Each time a memory location is referenced, the values of the word and block states are compared. The result of an access to a memory location can have one of three outcomes: hit, fmiss, or imiss (see Table 1 for definitions); these outcomes can be at variance when tracking coherence with word and block granularity. When these outcomes are different, we record that false sharing has occurred.

When word granularity coherence is used, only true sharing and compulsory misses can occur because actions on other words have no impact. This section focuses on comparing word and block states for each shared memory reference to determine the number of misses caused by false sharing.

Although in these simulations the memory transactions occur instantaneously, in the real world there are different penalties depending on the type of coherence operation/miss processing that must occur. Note that when necessary, we present the results of simulations incorporating realistic timings (e.g., Table 5). For example, the data may not be in the cache at all (requiring a fetch), or may be present but in the wrong state (requiring a coherence operation but no data transfer). To differentiate between the various memory system operations, we distinguish between two types of miss occurrences: fetch misses which require loading a word/block into the cache, possibly causing an invalidation of other copies of the block (**fmiss**); and an invalidation miss (**imiss**) caused by a write to a word/block in the shared state, requiring an invalidation operation. A cache reference can have one of three outcomes (**hit**, **fmiss**, **imiss**). If coherence is tracked at independently for words and blocks, an access to the cache can have one of the three outcomes at each coherence granularity, leading to 3^2 or 9 different outcomes. For example, a write reference might be a hit if tracked at word granularity, but be a miss for block level coherence, which we would refer to as a false sharing miss, because a preceding reference to another word in the block caused a miss to occur at the by changing the coherence state at block level, which would not have occurred with word coherence granularity. Appendix C shows examples of how each of the different cases can occur.

By our definition, a true sharing fmiss or imiss is one which occurs for both granularities of coherency. False sharing misses are those which occur when the state of the word using block coherency results in a coherency operation that would not have happened with word granularity. Three types of false sharing reference downgrades are possible: $hit \Rightarrow imiss$, $hit \Rightarrow fmiss$, and $imiss \Rightarrow fmiss$, where the first outcome is word granularity coherence and the second result from whole block coherence. Three corresponding upgrades (useful prefetches/state changes) in misses are also possible, resulting from the prefetching effect of block transfers, of which one type still requires a coherency operation ($fmiss \Rightarrow imiss$). The other three possible outcomes are those in which the type of miss or hit is identical for both word and block coherence, which are classified as true sharing hits or misses.

Table 3 provides a breakdown of the six types of sharing misses plus private/cold start misses on average for our workloads, distinguishing the cause of the misses and the comparison of the state of word granularity coherence vs. block granularity (e.g., hit to fmiss). It shows that false sharing fmisses begin to dominate true sharing fmisses for our workloads (on average) once blocks are bigger than 16 bytes, which would seem to disagree with the results of [TLH94]. However, most of our workloads (Table 6, and Figure 16 in Appendix D) individually agree with the results of [TLH94].

The extent to which the prefetching effect of larger blocks aids in turning misses into hits (which could be considered false sharing hits) can be observed by the decrease in miss ratio from the 1.0 normalized value as the block size increases. The two workloads with the heaviest fraction of false sharing misses (**pverify** and **topopt**) contain blocks exhibiting the classic example of false sharing, which is caused by arrays which have elements exclusively accessed using the processor ID as the index. These references would be unshared except they are grouped together into blocks, which results in the classical false sharing ping-ponging pattern. The workloads with the largest concentration of false sharing fetch misses ($hit \Rightarrow fmiss$) also show a large component of false sharing invalidation misses (hit to imiss). Conversions (upgrades) of misses to imisses occur so infrequently that they are not visible on this scale. However, significant downgrades of imisses to fmisses can be seen in **mp3d**, **pthor**, and **topopt** (Table 6 in Appendix D).

Data cache fetch miss ratios for the multiprocessor workloads fall far more slowly with an increase in the block size than the data cache miss ratios for uniprocessor workloads. Figure 3 shows a comparison of the infinite data cache fetch miss ratios for all workloads, the geometric average of these fetch miss ratios, and the DTMRs (design target miss ratios) for a 32 Kbyte data cache [Smi87] over a range of block sizes. As the block size increases, the rate of decrease of the miss ratios slows down, basically staying flat with 128- to 512-byte blocks.

Even in the region where false sharing is not a dominant effect (4 to 16-byte blocks), we can see that the workloads show less improvement with increasing block size than for uniprocessor DTMRs. Beyond 16 bytes the improvement in fetch miss ratio shows rapidly diminishing returns. In some cases, increasing the block size matches the prefetching effect of larger blocks with counteracting increases in false sharing fetch misses, keeping the

Breakdown of Misses, Normalized to 4–Byte Blocks											
Workload	Block Size	Total Misses	True Sharing				False Sharing				
			Fetch Misses	Inv. Misses	Cold Start Misses	Total	hit–> fmiss	hit–> imiss	imiss–> fmiss	fmiss–> imiss	Total
Average	4	1.000	0.423	0.256	0.321	1.000	0.000	0.000	0.000	0.000	0.000
	8	0.874	0.300	0.162	0.180	0.642	0.132	0.095	0.003	0.002	0.232
	16	0.854	0.228	0.108	0.104	0.440	0.256	0.150	0.004	0.004	0.414
	32	0.916	0.186	0.082	0.062	0.330	0.401	0.176	0.006	0.004	0.587
	64	0.992	0.161	0.069	0.038	0.268	0.526	0.187	0.007	0.005	0.725
	128	1.028	0.144	0.062	0.024	0.230	0.602	0.183	0.008	0.005	0.798
	256	1.114	0.119	0.058	0.014	0.191	0.710	0.199	0.009	0.005	0.923
	512	1.193	0.098	0.055	0.008	0.161	0.788	0.229	0.010	0.005	1.032

Table 3: Breakdown of workload misses into various types of false sharing and true sharing misses for 16 processors, normalized (ratio of miss ratios) to total (fetch, invalidation, cold start) misses for 4-byte blocks, arithmetic average over all workloads. The notations under the false-sharing heading, such as **hit->fmiss**, indicate the fraction references that are hits with 4-byte block coherence that become fetch misses with larger block sizes.

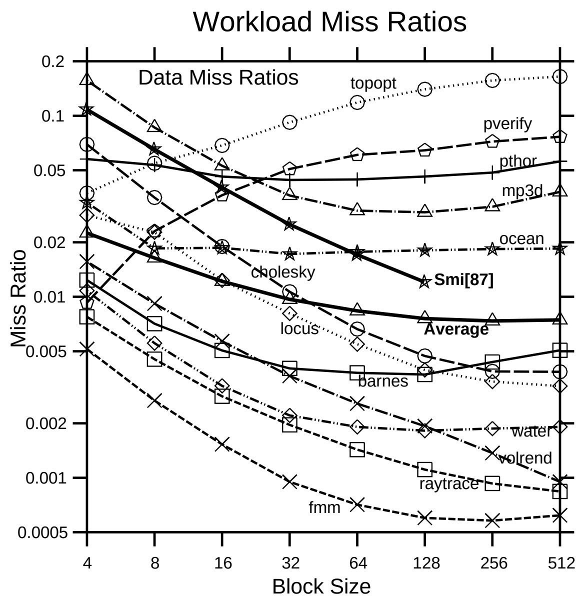


Figure 3: Data cache fetch miss ratios compared with 32 Kbyte data cache design target miss ratios from [Smi87]. The **average** value is computed using a geometric average.

overall number of misses fairly steady (**ocean**, **pthor**). Some of the workloads show a U-shaped curve, with misses hitting a minimum near 64 bytes (**barnes**, **water**, **mp3d**). Two workloads have the minimum number of misses with 4-byte blocks and explode with false sharing misses (**pverify**, **topopt**). Some workloads have little problem with false sharing and show continuing improvement with larger blocks (**cholesky**, **raytrace**, **volrend**).

In most SMP workloads, compulsory misses and true sharing are the cause of most of the misses and thus most of the bus traffic. However, as will be demonstrated, most of the words within a block are not accessed between invalidations to that block. This results in a large amount of unnecessary traffic from moving the unused words around, which we refer to as *dead sharing*. The next section (Section 4.1.2) examines the variation in the make-up of bus traffic as the block size increases. Even if the misses decline with block size, the traffic increases because the words within the larger block are not properly exploited, even for uniprocessor caches [RS99c]. Our results in Section 4.3 examines data utilization in a block (spatial-processor locality) while it resides in a cache.

4.1.2 Dead Sharing

In an attempt to measure the impact of large block sizes on bus utilization in an implementation independent manner, we use the bus traffic metric. Each transaction transmits a 4-byte address across the bus plus (when appropriate) some number of data bytes. The bus traffic consists of fetches (address + fmisses $\times$ block size) and invalidations (consisting only of the fixed overhead to transfer an address over the bus, since no data transferred is required). This is a reasonable estimate for bus utilization for split-transaction busses. Figure 4 shows average bus traffic for infinite caches with 4- to 64-byte blocks (relative to 4-byte block traffic), broken down into 5 main types of traffic: private traffic, global unshared traffic, invalidation signals (address transfer only), active shared traffic (truly utilized) and dead shared traffic.

The dead shared traffic is determined by analyzing which words in a shared block have not been accessed at the time the block is invalidated. The active shared portion of the shared traffic consists of the words that were

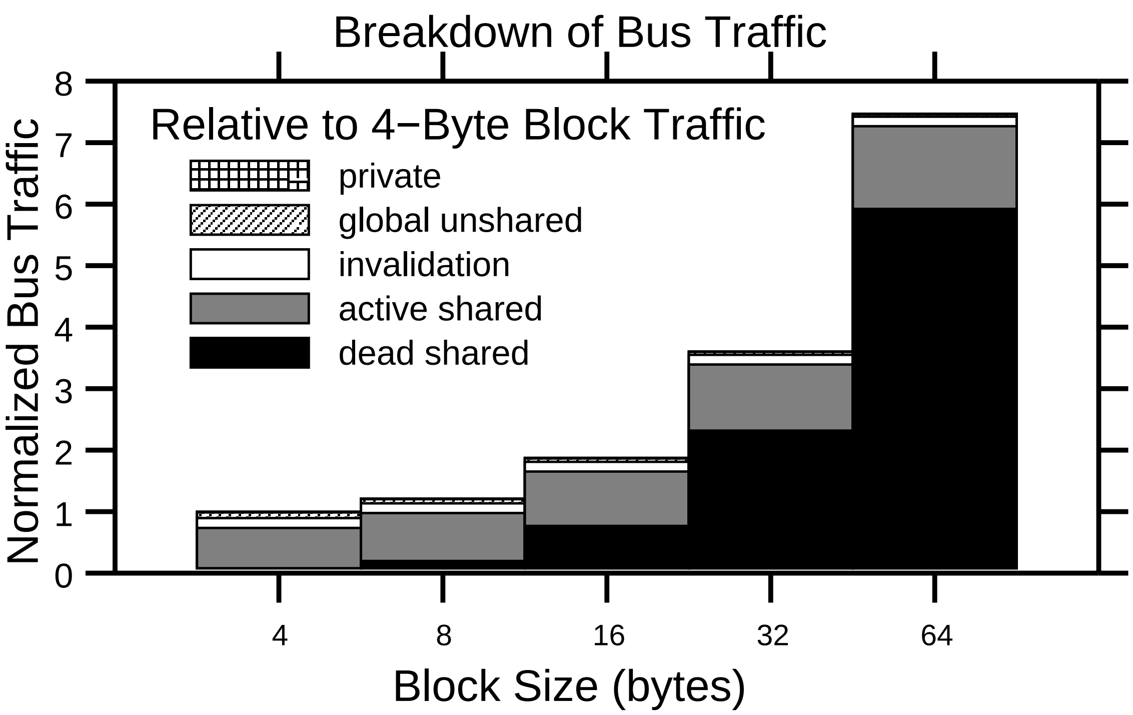


Figure 4: Breakdown of workload average bus traffic, normalized to 4-byte block traffic.

actually referenced before the block was invalidated. The breakdown shows that traffic from private blocks is relatively insignificant (from 1.17 percent for 4-byte blocks to 0.10 percent for 64-byte blocks on average) and traffic from global unshared blocks starts at 9.3 percent and declines to 0.5 percent for 64-byte blocks. Dead sharing traffic causes about 41.0 percent of the traffic with 16-byte blocks and grows very rapidly with larger block sizes. The traffic approximately doubles for each increase in block size beyond 64-byte blocks, reaching 54.3 times 4-byte block traffic when 512-byte blocks are used. The active shared traffic increases much more slowly than dead shared traffic. Increases in the active shared traffic are due to the incorporation of global unshared data into shared blocks as the block size increases, so that global unshared references are turned into active shared references, causing more active shared traffic. Dead sharing traffic hits 79.3 percent of total traffic with 64-byte blocks and reaches 95.1 percent with 512-byte blocks. This indicates that there is much room for enhancing the operation of shared memory systems.

Dead sharing traffic results from both false and true sharing that causes a block to be invalidated before all the words within the block can be utilized. The next section looks at the access patterns of distinct words within the blocks to understand the cause of this dead sharing.

Note that when trying to establish statistics like bus performance for a realistic system, there would be a number of parameters to consider. For example, bus utilization is a better metric than bus traffic for measuring how close to saturation the bus is. In such a case, implementation dependent issues must be considered, such as bus width, actual transaction overheads, bus pipelining, memory latency, split vs. non-split transaction, etc. Bus utilization and saturation issues are beyond the scope of this paper but are considered in [RS99a]. Assuming a one cycle address transfer time and a split-transaction bus, the bus traffic information in Figure 4 shows a reasonably good approximation of relative bus utilization between various block sizes. In a memory system that does not support split-transactions, the bus would be unusable during the memory latency period as well, which would cause a dramatic change in the relative bus utilization from what is displayed here.

4.2 Granularity of Sharing

When the set of words being written to a block by one processor shows little correspondence to the words read by other processors, there is a strong indication that false and dead sharing are a problem. For example, if generally one word in a block is the target for all writes to the block, but many of the other words are read-only after initialization (as occurs in the **GlobalMemory** (or similarly named) data structure used for global shared variables in many of our workloads), most of the data in that block is

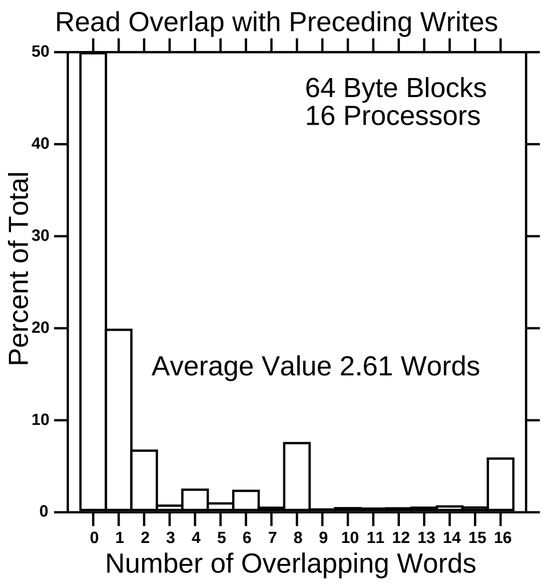


Figure 5: Read-write sharing during invalidation intervals, average of all workloads.

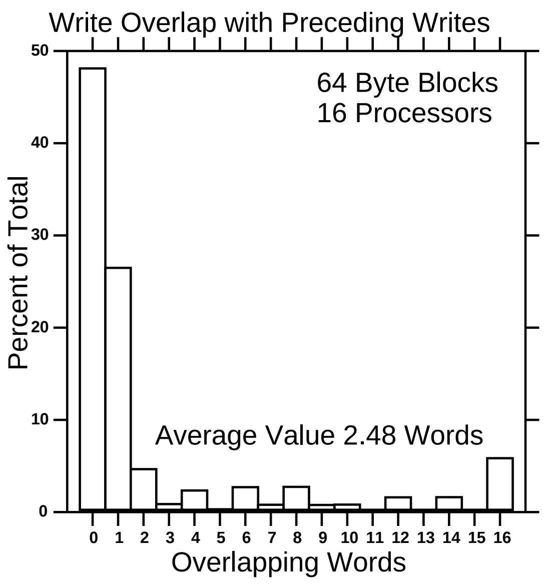


Figure 6: Write-write sharing between invalidation intervals, average of all workloads.

needlessly invalidated almost every time a write occurs.

Figure 5 shows a histogram of the number of reads that overlap preceding written words from the same invalidation interval for 64-byte (16-word) blocks, averaged over all workloads. Roughly 50 percent of invalidated cache blocks have no overlap of the words read in the second phase of an invalidation interval with the series of writes that began the invalidation interval, meaning that none of the updated information was accessed before another cache miss occurs for the processors reading the block. Approximately 20 percent of blocks have a read-write overlap of only a single word. The number of updated words read before an invalidation rapidly falls off, but with significant components for 8 and 16 words. These statistics demonstrate that half of the invalidations are caused by updates to words which are not subsequently read by other processors before the blocks are again invalidated, fully wasting the information transfer. A large fraction of those blocks which do read updated words only read a single word before invalidation. Increasing the block size affects the degree of overlap by increasing the likelihood that writes by different processors prematurely invalidate information in the block, which causes the average overlap to peak at 2.7 words for 128-byte blocks and drop with larger block sizes (Table 7 in Appendix E), indicating a massive waste of bus traffic.

Another method to examine how words within blocks are shared between processors is to measure whether writes by different processors to the same block happen to occur to the same set of words (write-write sharing granularity). Figure 6 shows a breakdown of the average case for 64-byte blocks, calculated by logically ANDing the vector of words written during successive invalidation intervals when the writers have different processor IDs. Almost 50 percent of the succeeding intervals which have different processors writing show no overlap in the set of words written. Of the remaining blocks which have at least one written word in common, the next largest value shows an overlap of only a single word. The histogram indicates a bifurcation of access patterns to blocks: either succeeding processors accessing a block have very little write sharing, or share a large fraction of the block (the 16 word component is somewhat prominent). Figures 5 and 6 indicate that a coherency protocol that could adaptively choose large or small granularity for enforcing coherence could be very successful in reducing coherency traffic.

4.3 Spatial Locality

A typical way to measure spatial locality for uniprocessors is to examine the change in the miss ratio as the block size varies. A more insightful metric for multiprocessors is our characterization of the read and write references to blocks between invalidations. A type of spatial locality (which we refer to as *processor-spatial locality*) can be determined by measuring the span and number of distinct words (defined in Table 1) referenced in a block while the block is valid in the cache, regardless of the interleaving read accesses by other processors. For each invalidated block, the words within the block are examined to see how well they were utilized. If too many of the fetched words are not used, bandwidth is wasted in the

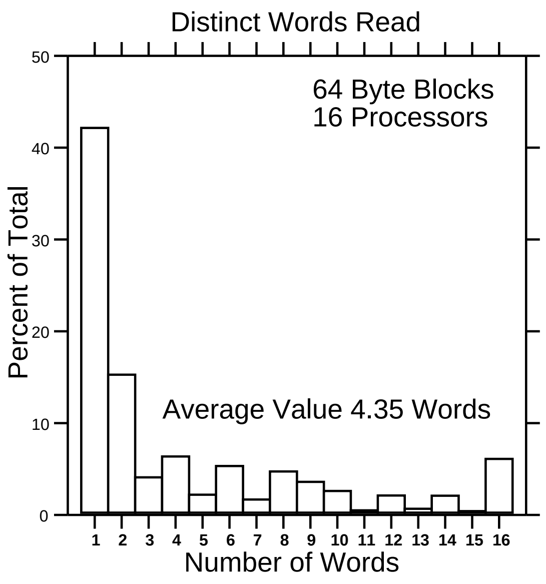


Figure 7: Distinct words read before invalidation, average of all workloads.

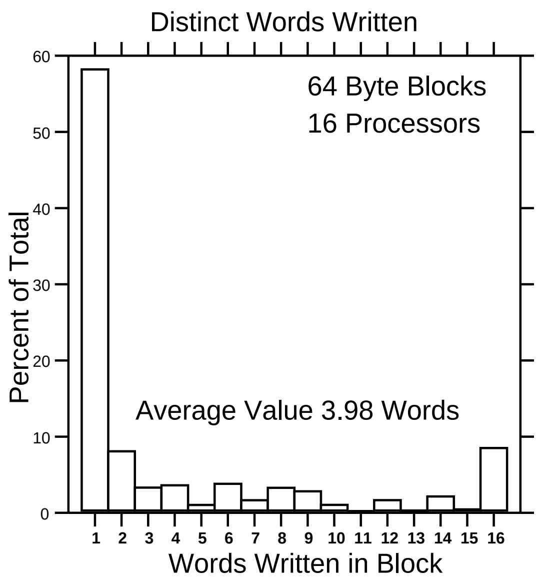


Figure 8: Distinct words written during an invalidation interval, average of all workloads.

form of dead sharing. Likewise, if too many words in the invalidated block are not subsequently updated, valid data was needlessly canceled and likely will have to be re-read, also a waste of bus bandwidth. An increase in block size increases the likelihood that a remote write to unrelated data can unnecessarily invalidate a block from the cache, causing dead sharing.

To provide a more detailed view of spatial locality, Figures 7 and 8 show the number of distinct words read within a block and the number of words written to a block while the block is in the cache, respectively, for 64-byte blocks, averaged over all workloads. This shows that the plurality (42 percent) of the blocks are invalidated by another processor before more than one distinct word is read, and only a single word is written 58 percent of the time during an invalidation interval. Except for single word blocks that have 100 percent usage (due to demand fetching), the percentage of blocks with only a single word read or written before a block is invalidated holds in a narrow range over block size, ranging on average between 37.6 and 45.8 percent for reads, and between 48.5 and 64.0 percent for writes (Figure 19 in Appendix E). The data shows that not much of the information in a block is used in any manner before another processor interferes, either by causing an invalidation of the data (in the case of a remote write), or by reading the block (placing it in the **shared** state), forcing the next write to cause an invalidation miss.

These figures indicate that it is not often that a whole block need be invalidated, since frequently only a single word is written before another invalidation occurs, which could be caused by a remote write, or a remote read followed by a local write. For many of the workloads, the histograms of the number of distinct words read/written (Figures 7 and 8) are very similar in composition to the span of the words read/written (Figures 17 and 18 in Appendix E), indicating a great deal of spatial locality to the portions of the block that are used. To understand the underlying cause of the poor block usage, we next examine the memory reference patterns that cause dead and false sharing to occur.

4.4 Examination of Problem Blocks

Many of the dead and false sharing problems revealed in the previous sections can be directly linked to poor programming style or ignoring the role of the cache in shared memory systems. Although caches are designed to be invisible to the programmer, poor data placement can have a large effect in reducing system performance.

Table 4 shows statistics about the 10 worst behaving blocks (based on the number of fetch misses) for each of our 12 workloads. For each workload, we specify the fraction of actively shared memory these blocks occupy, the fraction of total shared references to these blocks, the fraction of false and true sharing fetch and invalidation misses for which these blocks are responsible, and a classification of the reference patterns of the words within the blocks. The categorization of the words within the blocks are partially based on reference patterns classified in [GW92, CBZ95]. The 10 worst blocks are (on average) responsible for a good deal of the false sharing misses as

Categorization of Words Within the 10 Worst Blocks																	
Program	10 Worst Blocks						Reference Types (percent of words)										
	% Shared		% Fmisses		% Imisses		Private				Shared						
	mem	refs	False	True	False	True	un	pl	pro	prw	hl	ll	ro	rm	mi	br	rw
barnes	1.62	4.7	22.7	18.7	11.1	26.7	14.4	0	0	0	0.6	31.9	0	1.9	1.2	48.1	1.9
cholesky	0.06	11.4	75.2	6.1	37.3	0.5	6.2	0	0	3.1	0	88.1	0.6	0.6	0	1.2	0
fmm	0.12	0.1	8.4	4.8	0.2	2.4	0.6	0.6	0	0	1.2	17.5	0	0	0	0	80.0
locus	0.03	8.2	53.0	1.9	20.3	5.9	30.0	6.2	1.2	20.6	1.2	8.8	2.5	0	10.0	14.4	5.0
mp3d	0.08	2.6	9.1	5.3	2.9	4.7	11.9	0	1.2	2.5	0.6	1.2	3.8	0.6	0	10.6	67.5
ocean	0.14	4.7	20.9	35.2	3.8	25.9	10.0	0	0	32.5	2.5	0	36.2	3.8	0	15.0	0
pthor	0.03	30.5	63.4	49.4	36.1	38.4	16.9	0	15.6	1.2	1.2	12.5	6.2	0	0	31.2	15.0
pverify	0.52	7.4	29.5	5.7	29.9	0.2	10.0	0	1.2	61.2	0	10.0	6.2	0	0	10.6	0.6
raytrace	0.05	1.0	56.4	52.2	67.6	83.8	46.2	0	8.8	0.6	1.2	10.6	7.5	0	0.6	15.0	9.4
topopt	1.27	18.0	65.2	8.0	92.3	7.6	10.0	0	0	2.5	0	0	0	0	45.6	33.1	8.8
volrend	0.06	9.8	50.5	5.6	0.1	18.3	70.0	0	0	0	1.2	10.6	1.2	3.1	0	8.1	5.6
water	1.04	0.9	100.0	26.6	0	0.7	7.5	0	0	0	1.2	91.2	0	0	0	0	0
Average	0.4	8.3	46.2	18.3	25.1	17.9	19.5	0.6	2.3	10.3	0.9	23.5	5.4	0.8	4.8	15.6	16.2

Table 4: Categorization of the word reference patterns of the worst behaving 64-byte blocks for each workload. The categories for classifying each word in the worst blocks consist of: **un**- unused, **pl** - private locks, **pro** - private read-only, **prw** - private read-write, **hl** - high contention locks ($>$ 3 seeking access), **ll** - low contention locks (3 or fewer processors seeking the lock on average), **ro** - shared read-only, **rm** - shared read-mostly (at least 75 percent of references are reads), **mi** - migratory (more than 6 uninterrupted references on average by each processor accessing it), **br** - broadcast (one processor writing, many processors reading), and **rw** - read-write (words that do not fall into the other categories).

well as many of the true sharing misses. The number of misses are far out of proportion to the number of memory references and the fraction of shared memory space these blocks occupy. The basic problem is that variables are placed (perhaps inadvertently during dynamic memory allocation) into the same blocks as variables or data structures with incompatible reference patterns (for example, arrays of private read-write variables that are accessed by processor ID, read-only variables next to frequently written variables, etc). Almost one-fourth of the words in these blocks are locks with low contention (i.e., not much competition for them), that in isolation would cause little problem, but interact poorly together because locks have poor processor-spatial locality of reference. Other problem words are **broadcast** words that cause false sharing misses when placed together (but are still likely to have a fair number of true sharing misses in isolation), and read-write variables that interact together poorly.

In Appendix B we provide a more detailed analysis of each of the workloads to determine the kind of data structures that are causing most of the problems. Here we present a summary of our attempts to restructure 4 of the workloads ourselves, and some programming hints to prevent such problems in the future.

4.4.1 Improvements

Four workloads (**barnes**, **pthor**, **topopt**, **water**) were chosen to be restructured in an attempt to improve program performance. We modified the workloads to repair problems observed in the worst ten (64-byte) blocks of each. The details of the changes made to the workloads, and the data structures associated with the 10 problem blocks for each of the workloads can be found in Appendix B.

When modifying the data structures involved with the worst 10 blocks, some of these changes carry over to other blocks not in the top 10. However, improvements to the worst 10 blocks could also worsen behavior in other blocks. For example, isolating pieces of data structures by placing them in their own blocks (by adding unused arrays of integers to pad-out the members), can cause misses to increase for many well performing blocks with the same data layout, causing lackluster improvement in the number of misses.

For most of the restructured programs, the number of instructions increases slightly. This is generally due to the extra pointer dereferencing required by isolating per-processor data in separate data structures. On average, the number of instructions increases by less than 1.5 percent, while reducing the number of misses by more than 20 percent (Table 5). Although it depends on how much of a problem the data miss ratio is to begin with, this ap-

Data Restructuring to Reduce Coherence Misses (64-byte Blocks)					
Program	Infinite Cache, Single Cycle Memory			Realistic System Execution Time	
	Overall Miss Reduction	False Sharing Miss Reduction	Instruction Increase	16K cache	64K cache
barnes	12.0%	21.1%	0.0%	-1.4%	-6.7%
pthor	21.4%	53.3%	0.3%	-7.2%	-9.5%
topopt	31.9%	30.1%	3.9%	-42.3%	-46.0%
water	20.0%	99.9%	0.2%	-9.0%	-6.0%
Average	21.3%	51.1%	1.1%	-15.0%	-15.4%

Table 5: Results of rearranging data structures to reduce coherence misses, 64-byte blocks.

pears to be a reasonable trade-off. Using a multiprocessor timing simulator with 24 processor cycle memory latency supporting split transactions, 4-byte memory path (4 processor cycles per word), 4-byte memory addresses for each memory transaction, and 4 processor-cycle bus arbitration, we present (Table 5) the effects of the optimizations for two cache sizes (16K and 64K bytes per processor for 16 processors). Since the effects on the miss ratios reported in Table 5 are for infinite caches with single cycle memory accesses, effects such as capacity misses are not included. (The timing simulations, in the last two columns, are based on finite cache sizes.) When using small caches, the capacity misses can overwhelm the coherence misses, possibly worsening behavior if spatial locality is disturbed too much. The end result of the optimizations reduced execution time by approximately 15 percent. In the case of **topopt**, which spends most of its time waiting for memory, the spatial locality was increased at the same time that false sharing was reduced, leading to a tremendous increase in performance. In the next section, we present some programming hints; however, these optimizations must also be reconciled with the impact they have on spatial locality and capacity misses.

4.4.2 Programming Hints

Based on the detailed examination of the problem areas of our workloads, we provide here a distillation of the poor programming choices that lead to so many false sharing misses: high contention locks should be isolated from each other and from all other data; in many programs they are kept in arrays. Low contention locks should be placed with the data they protect. Some arrays (regardless of data type) are accessed using the ID of the processor as the index into the array; in some cases this results in a group of essentially private read-write variables being assigned to the same block, causing a large quantity of false sharing misses and dead sharing traffic.

Sometimes variables that appear to have true sharing misses can be restructured to eliminate almost all misses. For example, in **pthor** each processor accesses a particular shared variable only to increment the value. The value is actually only used in extremely rare cases (none that we observed during program execution), but the incrementation by each processor causes many true sharing misses. This variable can be restructured to isolate private copies for each processor, to be summed up when the value is actually needed. By examining program behavior more carefully using tracing and by programming with cache coherence in mind, significantly higher performance can be obtained.

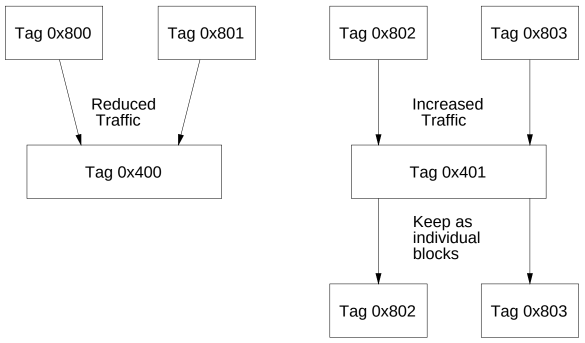


Figure 9: Heuristic algorithm: neighboring blocks are combined into a larger block when it reduces traffic, otherwise the blocks are left at their current size.

4.5 Proper Block Sizing

To demonstrate the performance improvement that can be obtained by reducing false and dead sharing, we use data collected from trace driven simulations of each program to find the best block size for each individual word in the memory space. Each program was simulated with block sizes from 4 to 512 bytes. For each shared word, we kept track of the address tag, the number of imisses and fmisses, and various other statistics. Using a simple greedy algorithm designed to minimize bus traffic, we demonstrate that a cache that supports multiple block sizes significantly outperforms all fixed-block systems.

The heuristic algorithm that is used to select the block sizes is designed to minimize bus traffic through the use of variable (static) size blocks; i.e., the block size choice varies over the memory space of the program, but any given word is assigned to a specific fixed block size for the entire program execution. Included as part of the bus traffic is a 32-bit (4-byte) address for each bus transaction (both imisses and fmisses) and the data transferred over the bus (fmisses only). Figure 9 shows the process by which blocks are evaluated for the best size. Starting with each word in the memory space that is used, neighboring blocks are combined if when combined they produce less bus traffic than when left as single blocks. When neighboring words have similar access patterns and it is useful to prefetch one while demand fetching the other, the traffic is reduced when the words (or blocks) are grouped into a single unit due to fewer address transmissions over the bus. When excessive traffic is generated due to false or dead sharing, the problem blocks are isolated by not combining them into larger units. The combining process continues until the maximum block size of 64 bytes is reached.

Figure 10 shows the fixed (uniform) block size simulation performance, normalized to the performance of our heuristic (the line across the lower two graphs at value 1.0; note the log scale on the vertical axis). The heuristic uses less traffic than any fixed block size for all workloads, sometimes as much as 47 times better in the most extreme case (**pverify**). On average it has 87.8 percent less traffic than a 64-byte fixed size block. At the same time, the number of misses is reduced by an average of 35.2 percent. In the worst case it is still within a factor of two of misses for fixed 64-byte blocks (**volrend**). Compared with 4-byte fixed size blocks, the heuristic has 70.4 percent fewer misses and 23.8 percent less traffic. Note that other heuristics are possible; for example, one could try to minimize the miss ratio rather than the bus traffic. Timing simulations would be required to determine which heuristic performs the best, but we believe that reducing traffic (while not increasing misses) on a shared bus system is a reasonable simple target, given that bus utilization is typically the bottleneck, and that bus traffic correlates with cache misses, and therefore CPU idle.

The block sizes chosen using our heuristic (the diagram second from the top of Figure 10) are most frequently 4-bytes and 64-bytes, with 8-byte blocks slightly less popular. That these two extremes are most popular is not surprising, based on the results from previous sections. Large blocks are best for shared regions with high processor locality; small blocks work best for regions in which there is a high probability that adjacent words are in use by different processors. Note, however, that in general there is a large variation between the optimal block sizes between the different workloads. We can also see that when the number of blocks of each size is multiplied by the size of the blocks, most words are still included in the bigger blocks (top of Figure 10). From the results shown here, we conclude that: (1) the use of variable block sizes permits the system to compensate for a mixture of false sharing and high processor-spatial locality; (2) alternately, it should be possible for the programmer to rewrite his or her code to avoid many false sharing situations (Table 4). Note that the method we have used for this analysis would generally be of very little use in a real computer system, since applying it would require that programs be traced and analyzed, and that each block of the program address space be tagged (or otherwise identified) with a block size. It might be possible to have the compiler do some static analysis, and associate block sizes with regions, but the effectiveness of that approach has not been considered here. The purpose of our analysis, rather, has been to identify a promising direction for improvement.

In a follow-on paper [RS99a], we use the results of this research to develop an invalidation-based cache coherence protocol that uses dynamically-sized subblocks for fetching and invalidation. By tracking the pattern of writes to a block between remote events to the block, the smallest subblock with a power-of-two number of words that contains the modified words is used as the subblock size. The subblock size is reevaluated occasionally, and adjusted to the most commonly measured value. Using variable subblock sizes, we find that our protocol outperforms a regular full block coherence protocol for all workloads, reducing the execution time by 35 percent (on average), as well as outperforming fixed size subblock protocols.

5 Conclusions

In this paper we have analyzed shared memory misses and bus traffic at three levels: in aggregate, statistically as words within blocks during the invalidation interval, and by examining special/bad cases in fine detail. The bulk analysis of misses shows that false sharing is generally not the largest fraction of the total misses for most workloads, being fewer than cold start and true shared misses. When analyzing the traffic caused by cache coherence, we do find that a significant problem is the fraction of bus traffic that is transferred between caches without being accessed, which we refer to as **dead sharing**.

Our analysis of invalidated blocks shows that typically only a small fraction of a block is referenced before it is invalidated. Generally there is little or no overlap between the regions of cache blocks updated by writing processors and read by other processors between invalidations to the blocks. Processors writing to the same block show very little overlap in most situations, but a great deal of overlap in a significant number of occasions. From this analysis

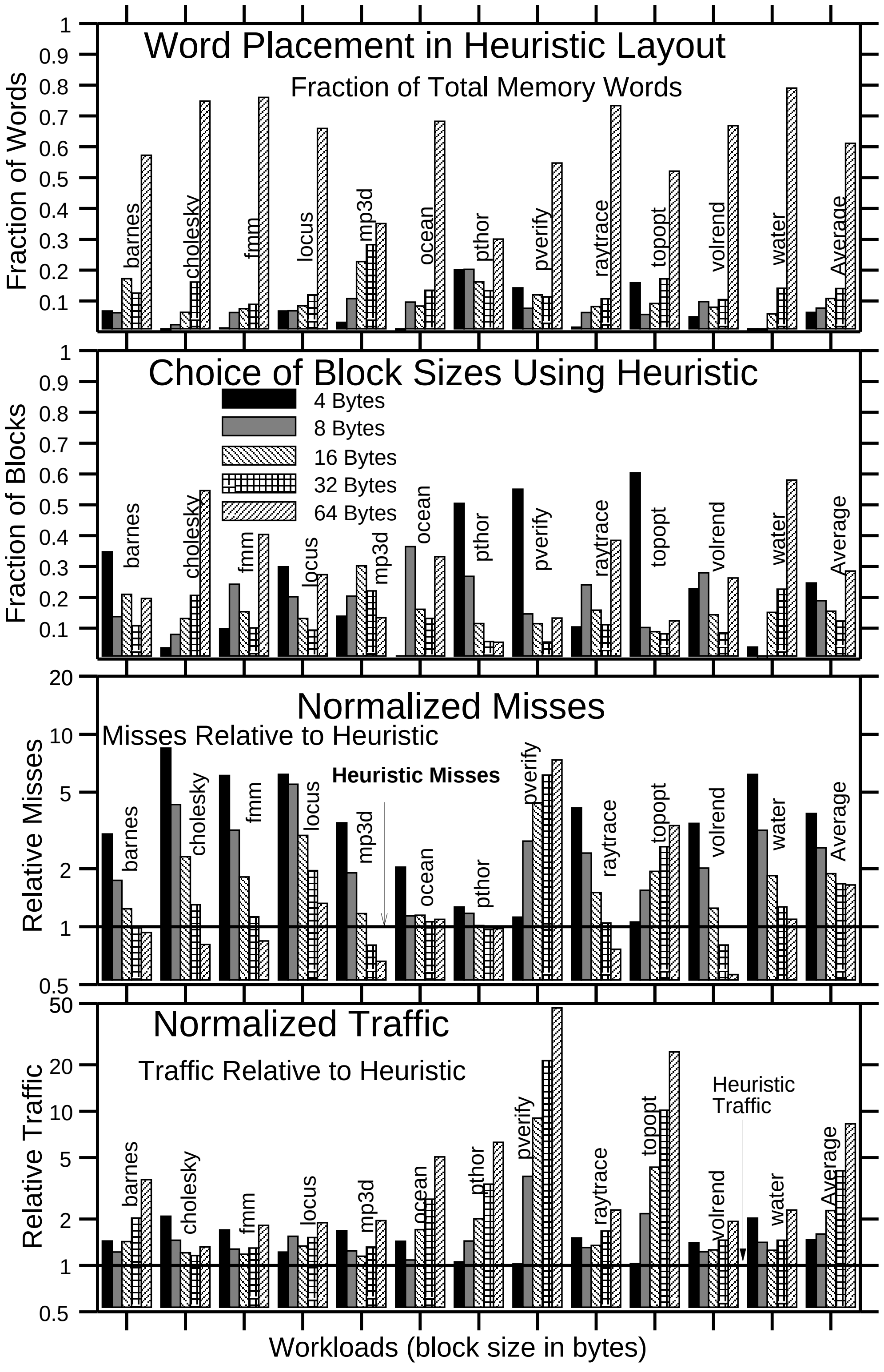


Figure 10: Normalized traffic and misses for fixed sized blocks normalized with respect to the variable block size heuristic and the choice of block sizes the heuristic uses. Note the log scale on the y-axis on the lower two graphs.

we believe a good case can be made for adaptively detecting the granularity of sharing within individual blocks and appropriately adjusting the portion of the block that is invalidated.

False sharing (and to some degree true sharing) shows a tremendous degree of concentration. The ten blocks with the highest number of misses from each workload contain close to half of all false sharing misses on average and a large number of true sharing misses and invalidations. These blocks generally take up a tiny fraction of the shared memory space and a small fraction of total data references. By looking at the reference patterns of each of the individual words within the offending blocks we found a large problem with arrays of locks and arrays of otherwise private words that exhibit classical false sharing. Another significant problem was frequently accessed read-only variables placed in proximity to write-shared variables. The concentrated nature of bad behavior indicates that a little attention to detail by the programmer would go a long way towards reducing misses and significantly improving performance; our efforts led to a 21 percent decrease in total misses, resulting in a 15 percent decrease in simulated execution time.

We examined a simple greedy algorithm heuristic which determined the best size block with which each word in main memory should be associated. Based on the results of this heuristic, we find that by using a variety of block sizes, bus traffic can be reduced a significant amount over 64-byte fixed size blocks while generally reducing miss ratios. Many of the best choices of block sizes for improving performance using our heuristic were 4- and 8-byte blocks (due to false and dead sharing), yet most of the data should be placed in larger blocks. This indicates that a cache that supports variable granularity fetching and invalidation (i.e., judicious use of subblocks) should greatly enhance program performance.

References

[AB95] Craig Anderson and Jean-Loup Baer. Two Techniques for Improving Performance on Bus-based Multiprocessors. In *Proc. First IEEE Symposium on High-Performance Computer Architecture*, pages 264–275, Raleigh, NC, January 22–25 1995.

[AG88] Anant Agarwal and Anoop Gupta. Memory-Reference Characteristics of Multiprocessor Applications under MACH. In *Proc. 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 215–225, Santa Fe, NM, May 24–27 1988.

[BH86] J. E. Barnes and P. Hut. A Hierarchical O(N log N) Force Calculation Algorithm. *Nature*, 324(6096):446–9, December 1986.

[BS93] William J. Bolosky and Michael L. Scott. False Sharing and its Effect on Shared Memory Performance. *USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 57–71, September 1993.

[CBZ95] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[CGHM93] David R. Cheriton, Hendrik A. Goosen, H. Holbrook, and Philip Machanick. Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: a First Experience. In *Proc. 1993 Workshop on Parallel and Distributed Simulation*, pages 159–162, San Diego, CA, May 16–19 1993.

[CM81] K. M. Chandy and J. Misra. Asynchronous Distributed Simulation Via a Sequence of Parallel Computations. *Communications of the ACM*, 24(11):198–206, April 1981.

[Dah95] Fredrik Dahlgren. Boosting the Performance of Hybrid Snooping Cache Protocols. In *Proc. 22nd Annual International Symposium on Computer Architecture*, pages 60–69, Santa Margherita Ligure, Italy, June 22–24 1995.

[DN87] S. Devadas and A. R. Newton. Topological Optimization of Multiple Level Array Logic. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-6(6):915–941, November 1987.

[DSR$^+$93] Michel Dubois, Jonas Skeppstedt, Livio Ricciulli, Krishnan Ramamurthy, and Per Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proc. 20th Annual International Symposium on Computer Architecture*, pages 88–97, San Diego, CA, May 16–19 1993.

[EJ91] Susan J. Eggers and Tor E. Jeremiassen. Eliminating False Sharing. In *Proc. 1991 International Conference on Parallel Processing*, pages I–377–I–381, August 12–17 1991.

[EK88] Susan J. Eggers and Randy H. Katz. A Characterization of Sharing in Parallel Programs And Its Applicibility to Coherency Protocol Evaluation. In *Proc. 15th Annual International Symposium on Computer Architecture*, pages 373–382, Honolulu, HI, May 30–June 2 1988.

[EK89a] Susan J. Eggers and Randy H. Katz. Evaluating the Performance of Four Snooping Cache Coherency Protocols. In *Proc. 16th Annual International Symposium on Computer Architecture*, pages 2–15, Jerusalem, Israel, May 28–June 1 1989.

[EK89b] Susan J. Eggers and Randy H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proc. Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, Boston, MA, April 3–6 1989. ACM.

[GS94] Jeffrey D. Gee and Alan Jay Smith. Analysis of Multiprocessor Memory Reference Behavior. In *Proc. 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 53–59, Cambridge, MA, October 10–12 1994.

[GS96] Jeffrey D. Gee and Alay Jay Smith. Evaluation of Cache Consistency Algorithm Performance. *Proc. Fourth International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 236–248, February 1–3 1996.

[GW92] Anoop Gupta and Wolf-Dietrich Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[HL90] Mark D. Hill and James R. Larus. Cache Considerations for Multiprocessor Programmers. *Communications of the ACM*, 33(8):97–102, August 1990.

[HP96] John Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan-Kaufmann, 2nd-edition edition, 1996.

[JE95] Tor E. Jeremiassen and Susan J. Eggers. Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations. In *Proc. Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, Santa Barbara, CA, July 19–21 1995.

[KB95] Murali Kadiyala and Laxmi N. Bhuyan. A Dynamic Cache Sub-block Design to Reduce False Sharing. In *Proc. International Conference on Computer Design: VLSI in Computers and Processors*, pages 313–18, Austin, TX, October 2–4 1995.

[LC86] G. C. Lie and E. Clementi. Molecular-Dynamics Simulation of Liquid Water with an ab initio Flexible Water-Water Interaction Potential. *Physical Review A (General Physics)*, 33(4):2679–93, April 1986.

[Lil93] David J. Lilja. Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons. *ACM Computing Surveys*, 25(3):303–338, September 1993.

[MDWS89] Hi-Keung Tony Ma, Srinivas Devadas, Ruey-Sing Wei, and Alberto Sangiovanni Vincentelli. Logic Verification Algorithms and Their Parallel Implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(2):181–189, February 1989.

[RG90] Edward Rothberg and Anoop Gupta. Techniques for Improving the Performance of Sparse Matrix Factorization on Multiprocessor Workstations. In *Proc. Supercomputing '90*, pages 232–41, New York, NY, November 12–16 1990.

[Ros88] Jonathan Rose. LocusRoute: A Parallel Global Router for Standard Cells. In *Proc. 25th ACM/IEEE Design Automation Conference*, pages 189–95, New York, NY, June 12–15 1988.

[RS99a] Jeffrey B. Rothman and Alan Jay Smith. An Adaptive Subblock Coherence Protocol for Improved SMP Performance. Technical Report UCB/CSD-99-10XX, Computer Science Division, University of California, Berkeley, Berkeley, CA 94720, October 1999. In preparation.

[RS99b] Jeffrey B. Rothman and Alan Jay Smith. Multiprocessor Memory Reference Generation Using **Cerberus**. Technical Report UCB/CSD-99-1054, Computer Science Division, University of California, Berkeley, August 1999. Available from http://www.cs.berkeley.edu/∼rothman/cerberus and also http://sunsite.Berkeley.EDU/NCSTRL.

[RS99c] Jeffrey B. Rothman and Alan Jay Smith. Sector Cache Design and Performance. Technical Report UCB/CSD-99-1034, Computer Science Division, University of California, Berkeley, January 1999.

[SGL94] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel Visualization Algorithms: Performance and Architectural Implications. *IEEE Computer*, 27(7):45–55, July 1994.

[SH91a] Jaswinder Pal Singh and John L. Hennessy. An Empirical Investigation of the Effectiveness and Limitations of Automatic Parallelization. In *Proc. International Symposium on Shared Memory Multiprocessing*, pages 25–36, Tokyo, Japan, April 2–4 1991.

[SH91b] Jaswinder Pal Singh and John L. Hennessy. Data Locality and Memory System Performance in the Parallel Simulation of Ocean Eddy Currents. In *Proc. Second Symposium on High Performance Computing*, pages 43–57, Montpellier, France, October 7–9 1991.

[SH92] Jaswinder Pal Singh and John L. Hennessy. Finding and Exploiting Parallelism in an Ocean Simulation Program: Experience, Results and Implications. *Journal of Parallel and Distributed Computing*, 15(1):27–48, May 1992.

[SHG92] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Implications of Hierarchical N-body Methods for Multiprocessor Architecture. Technical Report CSL-TR-90-439, Stanford, February 1992.

[SHHG93] J. P. Singh, C. Holt, J. L. Hennessy, and A. Gupta. A parallel adaptive fast multipole method. In *Proc. Supercomputing '93*, pages 54–65, Portland, OR, November 15–19 1993.

[Smi87] Alan Jay Smith. Line (Block) Size Choice for CPU Cache Memories. *IEEE Transactions on Computers*, C-36(9):1063–1075, September 1987.

[SS86] Paul Sweazey and Alan Jay Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proc. 13th Annual International Symposium on Computer Architecture*, pages 414–423, Tokyo, Japan, June 2–5 1986.

[Ste90] Per Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *IEEE Computer*, 23(6):12–25, June 1990.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical report, Stanford, June 1992. Report No. CSL-TR-92-526.

[TLH90] Josep Torrellas, Monica S. Lam, and John L. Hennessy. Measurement, Analysis, and Improvement of the Cache Behavior of Shared Data in Cache Coherent Multiprocessors. Technical report, Stanford, February 1990. Report No. CSL-TR-90-412.

[TLH94] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[WOT$^+$95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 22–24 1995.

A Workload Descriptions

The fundamental properties of the SPLASH workloads we use have been throughly described in [SWG92, WOT$^+$95]; here we provide a synopsis of the sharing behavior of all of our workloads based on descriptions in previously published works:

A.1 Barnes

Barnes [BH86, SHG92] measures the evolution of an N-body system under the influence of gravity using the $O(n\ log\ n)$ Barnes-Hut algorithm. The original three dimensional space is recursively broken up into eight equal sized pieces any time a space has more than one body in it. An octree is used as the data structure to represent this division process. At each time step and for each body, the force-calculation algorithm descends the octree, treating groupings of bodies as a single body if the grouping is far enough away, otherwise pair-wise interactions are examined for each leaf.

Before each new time step, the octree must be rebuilt from the new distribution of bodies resulting from the previous step. Writes occur during the body partitioning task and when the body positions are updated. During the force computations the sharing is read-only; these computations use the greatest proportion of execution time. In an attempt to balance the computation load among the processors, the tasks are roughly divided up between the processors in a manner that does not take locality in the data space into account.

A.2 Cholesky

Cholesky [RG90] performs a Cholesky factorization (find $L : LL^T$ = Input Matrix) of a sparse matrix, using the supernodal fan-out method. The input matrix is not a standard sparse matrix, but one that has been already reordered to reduce the amount of fill in the resulting lower triangular matrix L. Supernodes (set of columns with nearly identical non-zero entries) are used for efficiency purposes (including data distribution efficiency). The task size assigned to a processor is a supernode plus the set of all modification tasks the supernode performs on other supernodes. When the number of modifications required by a supernode becomes zero, the node is put on a shared task queue where it can start performing modifications to other supernodes. Sharing can occur when several processors performing operations on behalf of their own supernode tasks affect the values of the same supernode. Once a supernode has no remaining modifications to be performed on it, it is processed by a single processor, which may in turn affect other shared supernodes. Once a supernode has been completely processed, no further memory references occur to it.

A.3 FMM

FMM [SHHG93] is another N-body system studied over time (like **barnes**), but it uses the adaptive fast multipole method to simulate two-dimensional interactions. The data structures are similar to those in **barnes**, but the algorithm makes a single pass up and down the tree structure, rather than a pass through the tree for each body. Data is not distributed with any attempt to increase locality, so the data access patterns can be haphazard.

A.4 LocusRoute

LocusRoute [Ros88] is a VLSI standard cell router which attempts to minimize the cost of routing wires between cells. The tasks consist of wires, grouped into different geographical regions, which are assigning to available processors to route. When a processor finishes its tasks, it can look for wires to route in another processor's region. Wires are routed using a shared cost array of routing cells (eight bytes per cell) which keep track of the number of horizontal and vertical wires in each cell. These cells are not locked, which can occasionally lead to stale information. The cost array is the main area in which sharing occurs, but the degree of sharing is kept low by geographically assigning the routing. The other potentially shared data structures are the task queues and the data describing the wires' routes and pin positions, but these are not accessed as frequently as the cost array.

A.5 MP3D

MP3D [CGHM93] simulates hypersonic rarefied fluid flow through a rectangular shaped aperture. Molecules flow through a three dimensional space, mostly in a positive x direction. Occasionally the molecules collide with each other or with the boundary tunnel.

MP3D statically assigns molecules to processors, but there is no guarantee that molecules associated with a particular processor have any kind of locality within the 3-D space. On each time step, molecules are moved according to their motion vectors between unit-sized space cells. A collision takes place when two (or more) molecules occupy the same cell. The data sharing in **MP3D** occurs when the molecules of two different processors occupy the same cell. Any access to the array of space cells can also result in sharing.

A.6 Ocean

Ocean [SH91b, SH92] measures the effects of wind stress, planetary rotation, and friction on large-scale ocean movements. The body of water is confined within a cuboidal basin, and the role of eddy and boundary currents are examined. The simulation is run until the eddies and mean ocean flow reach a mutual equilibrium. On each time step, a set of spatial partial differential equations are solved, using an iterative red-black Gauss-Seidel multigrid solver.

Data consisting of numerous 2-D arrays are allocated among the processors in such a way as to minimize communications between them. This data is permanently assigned to each processor and only that processor writes to it. Read sharing may occur, but typically the data read by other processors was written during the previous iteration and is not modified during the current time step.

A.7 Pthor

Pthor is a parallel circuit simulator based on a variant of the Chandy-Misra algorithm [CM81]. Unlike many circuit simulators that keep all circuits synchronized to global time, **pthor** allows elements to advance at their own rates, based on input events. When input events occur, an element is placed on a task queue to be processed. Deadlock is possible using this algorithm and is detected and adjusted to allow further progress.

The circuits consists of wires (nodes) and logic elements. These form the primary data structures along with the distributed per-processor task queues, which are all potentially shared among the processors. Associated with each wire is an event list. Each element is assigned a preferred queue (associated with a processor), assigned when the circuit is read in, where it gets placed each time it is activated. The goal of data distribution is load balancing; there is no attempt to increase locality by assigning associated elements or wires to the same processor. Processors can steal tasks from other processors when their task queues are empty.

A.8 Pverify

Pverify [MDWS89] verifies the Boolean equivalence of two combinational logic circuits. The method for verification involves tracing backwards from a primary output to find the set of cubes that cause the output to be 1 (the ON set) or 0 (the OFF set). A cube is a subset of inputs with a certain value that determines the value of the output, regardless of the values of the inputs outside of the subset. Whenever a primary input is chosen for examination, two tasks are generated for the output value (one each for 0 and 1), which is put on a task queue. Once the ON and OFF sets have been found, they can be applied to the circuits under test to verify that they are equivalent.

The main data structure consists of information specifying all the inputs, wires, nodes, and gates leading to each primary output, which is referred to as a **cone circuit**. The data is organized in such a way that the cone circuits can be evaluated and simulated in parallel, since any of the wires and portions of the cone circuits can simultaneously be under test by different processors. The data structures use arrays indexed by processor ID to allow simultaneous evaluation. True data sharing mainly takes place in writing to the table of cubes and in maintaining the task queues, but the data organization causes major amounts of false sharing.

A.9 Raytrace

Raytrace [SGL94] is a three dimensional image renderer that uses ray tracing. The work space is split up using a hierarchical grid similar to octrees. A ray is traced backwards from a pixel in an image plane and generates a ray tree from reflections caused by contacts with objects. The problem is broken up into distributed task queues, consisting of contiguous pixels. The important data structures consist of the hierarchical grid, task queues, rays and ray trees, and the scene description. Data access patterns are not regular due to the unpredictable ray reflections.

A.10 Topopt

Topopt [DN87] (a.k.a. GENIE II) is a gate matrix layout package for automated layout synthesis of static CMOS, static NMOS, and DOMINO design styles. It uses simulated annealing to attempt to minimize the layout cost under the constraint of terminal locations and non-uniform transistor sizes.

At the beginning of the program, each processor is assigned a set of gates and signals in the array (referred to as a *window*). Each processor independently tests different arrangements within its window. Through the use of dynamic windowing, gates can occasionally be exchanged between windows. This allows the multiprocessor algorithm to generate high-quality solutions, similar in quality to uniprocessor solutions.

The main data structures consist of cells (gates) and the wire nets that connect the cells together. The cells are split among the processors, but all of the nets may be examined by each processor. To keep processors from interfering with each other (with respect to the nets) during window evaluation, each net has four arrays indexed by processor ID. This leads to massive amounts of false sharing during program execution.

A.11 Volrend

Volrend [SGL94] creates a series of two dimensional projections of a three dimensional object (the supplied input files contain a head) from different perspectives. It uses ray casting to render the object. Neighboring pixels from the image plane are assigned to each processor. The main data structures are voxels used to represent the three dimensional image, octrees to aid in aid in tracing the rays through the object, and image plane pixels. The data accesses are input dependent and unpredictable.

A.12 Water

Water [LC86, SH91a] is an N-body simulation of the forces and potential between water molecules. A box, which is large enough to hold the molecules, is the boundary for the system. To avoid calculating the $O(n^2)$ interactions between the molecules on each time step, a cut-off radius equal to half the length of the box is utilized.

The molecules are represented as an array of structures, one structure for each molecule. The structures contain a 3-D array describing the molecular layout, plus a smaller array representing center of mass information. Molecules, which are not necessarily adjacent in the simulated space, are statically assigned to processors. Data sharing can occur when the forces on pairs of molecules are calculated, but no more than half the processors involved in the problem will touch a particular molecule.

B Analysis of Problem Blocks

In Section 4.4 we noted that the worst 10 blocks (in terms of the number of fetch misses) contain many of the false sharing misses over the various workloads. In this appendix, we look at the actual data structures involved and describe what is causing the conflicts to occur. For four of the programs, we modified the source code to make improvements to improve software performance, the results of which are presented in Section 4.4.1. These programs were **barnes**, **pthor**, **topopt**, and **water**. For each of these select workloads, the modifications are also explained in this section. 64-byte blocks are used in all examples in this section.

B.1 Barnes

There are two main problems in **barnes**: (1) it uses arrays of locks; and (2) the **cell** structure, which is 78 bytes long, is spread over several blocks, in such a manner that the next data structure starts on the same block on which the previous one ended, resulting in 4 of the worst blocks. Although most of the locks are low contention (less than three processors competing for the lock on average), arrays of such locks cause massive false sharing. The **CellLock** array has 2000 entries, of which the first 50 entries result in four of the top ten worst blocks. The other two of the worst blocks are caused by the layout of the **GlobalMemory** data structure, by placing locks and by occasionally written variables in proximity to frequently read data.

To improve the program, the **GlobalMemory** structure was reorganized to group locks with the data they protect, and lock arrays were changed to isolate each lock in its own 64-byte block. Further attempts to make improvements by padding out the **cell** structure to fill an integer number of blocks slightly reduced the false sharing misses, but actually increased capacity misses and execution time for the 16K and 64K byte caches tested, due to the increased space required to store the data structures, so this optimization was eliminated.

B.2 Cholesky

Nine of the worst ten blocks consist of portions of an array of locks called **colLock**. The tenth block consists of three overlapping data structures: **work_tree** (an array of integers), the **GlobalMemory** structure, and the **TaskQ** structure. Two of these structures have locks, although only the lock associated with **TaskQ** is used.

B.3 FMM

Two of the bad blocks consist of a number of neighboring locks. The other 8 of the blocks have portions of an array (**local_expansion** in data structure **box**) which are of type complex, which has two double precision members and represent a complex number.

B.4 LocusRoute

Three of the worst blocks are the first three blocks of the **GlobalMemory** structure, which have problems because some of the frequently read members (which are generally read-only) are near members which are written often. In addition, there are several locks in close proximity to each other. Two other bad blocks consist of an array of type **TaskQueueLockRecord**, which has a lock and a short integer as members. Two blocks consist of an array of pointers which are frequently modified by various processors. The last 3 of the bad blocks are contiguous and hold an array of type **SegmentHeadSyncRecord**, which can all be manipulated simultaneously by different processors, leading to false sharing.

B.5 MP3D

Two of the bad blocks come from the **GlobalMemory** data structure, which consists of frequently read members mixed with frequently written data and various locks. The rest of the bad blocks consist of portions of the **Cell** data structure. **Cell** is smaller than a block and in the worst cases, portions of 3 data structures are placed in the same block, causing false sharing.

B.6 Ocean

The 10 worst blocks consist mostly of portions of the 4 dimensional array **q_multi**. The array layout is a function of the number of processors and the size of the problem being run. Because of the way it is created, each dimension of the (ragged) array requires pointers to be stored rather than calculated. In an apparent attempt to increase locality of reference, the order of allocation mixes dimensional pointers (exclusively read after initialization) with the read/write data in the **q_multi** array, causing a large amount of false sharing. One bad block consists of locks which neighbor each other.

B.7 Pthor

Six of the bad blocks in **pthor** consist of portions of the **globmem** structure, which consists of frequently read data mixed with read/write data and locks. We also discovered an interesting sharing issue: there are several variables (at least 5) which appear to be truly shared, but a closer analysis shows that in fact that the variables are mostly incremented, but very rarely (if at all) read to be used for other purposes, only for a **printf** statement. Clearly such a class of variables should be turned into an array accessed by processor, but in such a way as to minimize false sharing. Then, when the value is actually required, a loop could be used to accumulate the values from each processor.

The other 4 problem blocks are contiguous and consist of portions of an array. The array is of a data structure of type **ActList** which contains 3 pointers and a lock. There appears to be no locality to access between the various elements of the array, causing large amounts of false sharing.

To reduce the false sharing in **pthor**, the data structure **ActList** was padded out to 64 bytes to prevent interference between the locks in neighboring array members. A memory allocator was added to make sure that these data structures are aligned on 64 byte boundaries. Data structure **globmem** was slight rearranged to isolate two locks from each other and from other members. Additionally, a frequently read member **n** was separated from a frequently written member **DeadCount**.

B.8 Pverify

Among all the workloads, **Pverify** seems to be the example of how not to arrange data layout. One problem block contains an array of locks. Another one contains a two-dimensional array indexed by processor ID in the inappropriate dimension, in such a manner as to cause false sharing. The **WIRE** structure contains several arrays accessed by processor ID, resulting in 6 bad blocks. The other 2 problem blocks are caused by the manner in which data is allocated in shared memory: there are alternating **NODE** and **WIRE** structures, which do not work well when placed together.

B.9 Raytrace

The **gmem** structure is responsible for 2 of the worst blocks, arranging read-only, read/write and lock variables near each other. The other 8 of the ten worst blocks is caused by alternating shared memory allocations of **NODE** and **WPJOB** data structures. Because of the nature of this program, huge amounts of data are read compared to the amount written, and the written data is mostly isolated by processor, so there are few false sharing misses.

B.10 Topopt

Four of the bad blocks consist of the array **p_rows**, whose elements are generally migratory values (i.e., uninterrupted accesses by one processor for a while, then passed to another processor). The other bad blocks mostly consist of arrays (**right**, **nmax_height**, **cell_num1**, **cell_num2**, **net->nminp**, **net->nmaxp**), in which each element is written by only one processor (different processor for each array element), but read by several processors (broadcast reference pattern). Some of the blocks also contain portions of two arrays, such that the beginning of one array and the end of another array fall on the same block. False sharing misses dominate other sources of misses for all of the 10 worst blocks in this workload.

To attempt to fix the false sharing problems in **topopt** required creation of new data structures to aggregate information by processor, not by type or function. This required a fair amount of coding to restructure. For example, arrays called **right** and **left** were changed so that they became scalars members of a new data structure. This isolated data by processor and eliminated a good deal of false sharing. In fact, it increased spatial locality by grouping together data with good affinity, and showed tremendous effect in improving execution time for the real systems simulation, enhancing performance by reducing misses beyond the false sharing misses eliminated.

B.11 Volrend

Two of the bad blocks are caused by overlaps between three data structures and the reference patterns to them. Two of these data structures, **invjacobian** and **invinvjacobian**, are initialized by a number of processors writing to the same words at the same time, which appears to be a mistake in the program. Two other bad blocks are caused by mixtures of synchronization variables (both locks and barriers). The other blocks have portions of a large two dimensional array of integers. The first element of each row of the array (also the only element on each row that is referenced) is written and read by a number of processors, causing a large number of true sharing misses as well as significant dead sharing.

B.12 Water

The problem with the ten worst blocks in water is caused exclusively by a large array of locks neighboring scalar locks in the **GlobalMemory** data structure. We repaired this problem by allocating a block for each lock, largely eliminating false sharing from the program.

C Word vs. Block Coherence

This appendix provides several simple examples of how it is possible to account for a miss as a false sharing miss or what is meant by the terms *upgrade* and *downgrade*. In each example, we show an initial state that is the same for both word coherence/transfer size as it is for block coherence (at least for the words with which we are concerned). In each case, an initial reference occurs which changes the state. The second and third references are the important ones for each example, as they demonstrate the kinds of misses we wish to explain. All examples implicitly use sequential consistency.

For each of Figures 11-15, the word and blocks states are one of MESI states as defined in Section 4.1. Although a block's coherence state represents the state for all the

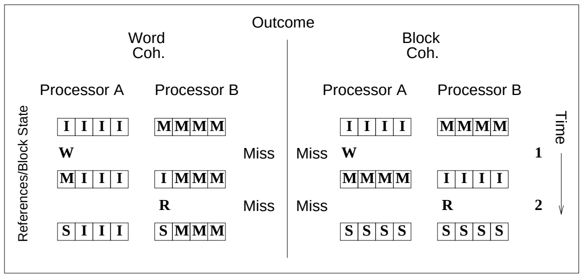


Figure 11: True sharing miss.

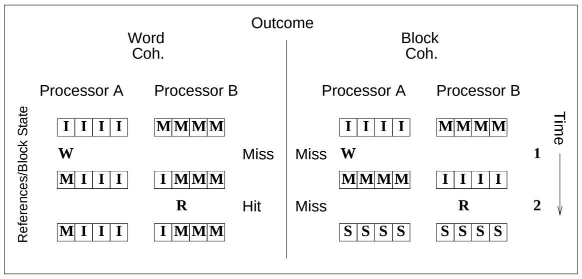


Figure 12: False sharing miss downgrade.

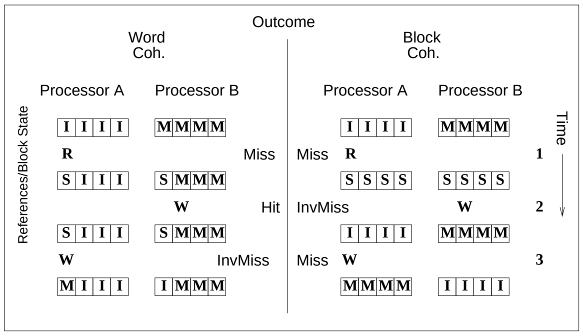


Figure 13: False sharing invalidation miss downgrades.

words within the block, we show the state of each word under block coherence to minimize confusion.

Figure 11 shows an example of a true sharing miss. Under both word and block coherence, the second miss occurs because the word in question is actually shared between the two processors.

False sharing misses occur when a word's coherence state is affected by references to other words in the block, so that the word is found in a different state under word and block coherence when the next reference to the word occurs. The state can be affected in two ways: the state can be worse than what is expected under word coherence (i.e., a *downgrade* has occurred), or it can be found in a better state than expected (an *upgrade*) has occurred. Keep in mind, we are not actually interested in the word's coherence state per se; rather we are interested in whether a preceding reference to another word in the block causes a data transfer or a coherence action when the word with which we are concerned is accessed. For example, a read to a valid word in the cache requires no coherence activity if the word's state is shared, exclusive, or modified. A write causes no externally observed coherence actions if the word's state is exclusive or modified. A write to a shared word causes an invalidation transaction; a read or write to an invalid word causes a data fetch.

A read downgrade is said to have occurred when a word is found to be invalid (because of actions performed on other words under block coherence) when it would be in a valid state with word coherence; a write downgrade (with two levels of seriousness) occurs when the state is either invalid (causing a fetch and invalidation) or shared (causing an invalidation) when modified or exclusive was expected. However, the memory system operation required for a write to a shared location is less than required when a read or write to an invalid word occurs (under write-allocate), so we distinguish between invalidation misses and fetch misses.

Figures 12 and 13 show the three different downgrades possible. Figure 12 exhibits how a reference to another word in the same block by another processor causes a different outcome for the reference to the word by processor B. The word was in a modified state, but a remote (reference by another processor) write causes the word to be found in an invalid state. This type of miss is generally the most common of the false sharing downgrades observed in our simulations and is the most serious.

In Figure 13, the read by processor A causes all the words' states to become shared under block coherence, causing an invalidation miss by processor B, yet under word coherence it would have been a hit. A write by processor A causes yet another miss, whereas under word coherence it would have caused only an invalidation, but no data transfer. So in this one figure, there are examples of two kinds of downgrades: hit to invalidation miss, and invalidation miss to fetch miss.

There are also three corresponding types of upgrades. Figure 14 shows an upgrade caused by a beneficial use of block coherence/fetching. A block fetch acts as a kind of prefetching in uniprocessors as well as for well behaving blocks in multiprocessor caches. By reading all the words in a block at once, a number of misses can be eliminated for subsequent accesses to the block under the right circumstances.

Block coherence can also cause less dramatic improvements in word states. When a read to a block that is present in another cache but invalid in the local cache occurs, the whole block is fetched and all the words are placed in the shared state (read by processor A in Figure 15). A subsequent write by processor A to another word in the block puts the entire block in the modified

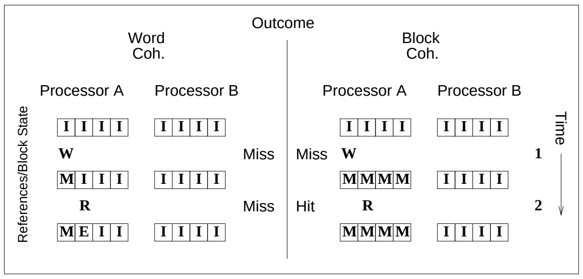


Figure 14: False sharing hit (prefetch upgrade).

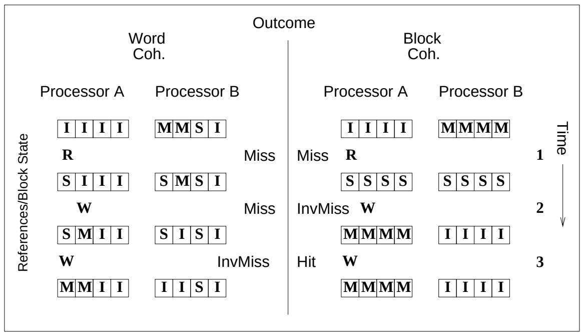


Figure 15: False sharing upgrades.

state, so that further writes by processor A are cache hits. Under word coherence, the write to another word only affects that word's state, so that the second write causes an invalidation miss. The prefetching effect of block coherence can work for states as well as for the actual data involved. So under these circumstances, block coherence outperforms word coherence. However, as found in the main portion of this paper, once the block size is sufficiently large, the benefit (in terms of miss ratio) of using blocks for coherence and data transfers rapidly diminishes.

D Breakdown of Data by Workloads

Table 6 shows the categorization of the types of misses for each individual workload, the average can be found in Table 3. Figure 16 shows the same information in graphical form. The total misses consist of both fetch misses and invalidation misses. In a real system, invalidation misses are less serious than fetch misses, but under sequential consistency, still cause the processor to stall while acquiring exclusive access to a block. For each workload, the number of misses is normalized to the total number of misses for 4-byte blocks. The misses are broken down into true sharing fetch misses (ts fmiss), true sharing invalidation misses (ts imiss), private and cold start misses, false sharing fetch misses (fs hit to fmiss), false sharing hit to invalidation miss (fs hit to imiss), and false sharing invalidation miss to fetch miss (fs imiss to fmiss).

As can be seen, for most workloads the number of cold start and true sharing misses exceed the number of false sharing misses. However, the four workloads with bad false sharing behavior (**barnes**, **pthor**, and especially **pverify** and **topopt**) cause the average over all workloads to have the majority of misses caused by false sharing starting in the 16- to 64-byte block region and beyond.

E Expanded Information on Sharing Granularity

This section contains figures and tables that have interesting content, yet were moved out of the main section due to space constraints. Most of the data is related to the information presented in Sections 4.2 and 4.3.

When a write occurs to a block in the shared state (presumably to update a shared variable), all copies of the block in other processors' caches must be invalidated. Before another remote read occurs, other words in the block may be modified as well. Other processors interested in the data must then re-read the block. Table 7 shows the average number of the updated words that are read before a block is invalidated (read-write sharing granularity, the bold entry is expanded into a histogram in Figure 5). In most of the blocks that are shared, we do not observe much of an overlap of local writes with remote reads during an invalidation interval, causing the average number of overlapping words to be on average rather small. Only programs that show a tendency to migrate data objects between processors (such as **water**) have a reasonable number of overlap of reads and writes. This value is an indication of the degree that false and dead sharing are occurring in the workloads. Notice that the amount read-write sharing is 100 percent for 4-byte blocks, in the situation where only true sharing can occur. With bigger block sizes, the number of overlapping words increases, but not nearly as fast as the block size. Once the block reaches a certain size, the amount of overlap starts decreasing due to the large probability that several unrelated data structures are placed in the same block and they really start interfering with each other.

Table 8 shows the degree of overlap between sets of writes to a block by different processors during succeeding invalidation intervals (when the writing processor changes). Workloads that show a large degree of false sharing have a extremely low degree of write overlap (**barnes**, **ocean**, **pthor**, **pverify**, **topopt**), but generally the amount of write overlap within a block is fairly low. Water has a fair degree of write-write overlap because the most typical memory reference pattern shows

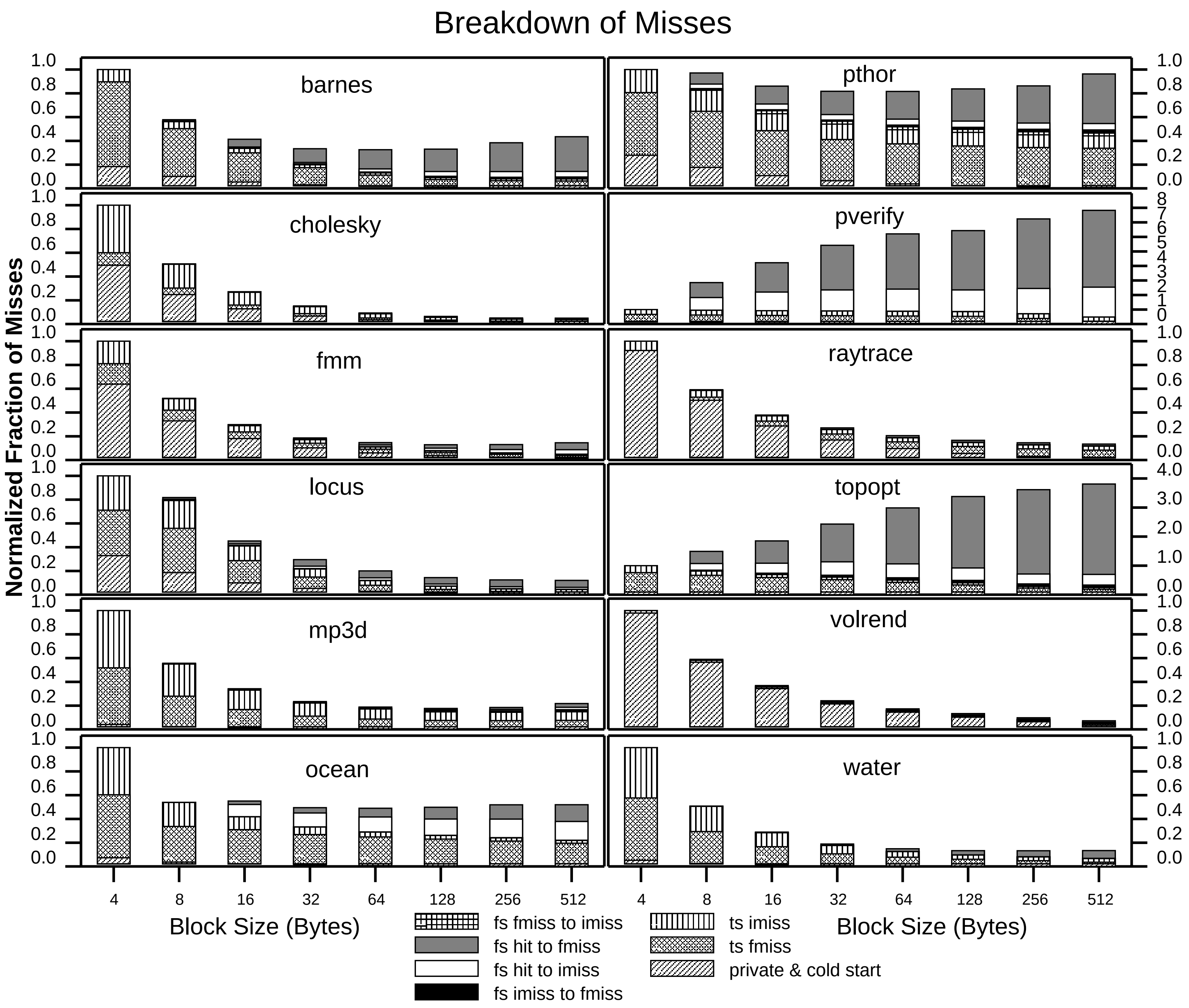


Figure 16: Breakdown of workload misses into various types of false sharing and true sharing misses for 16 processors, normalized to misses for 4-byte blocks.

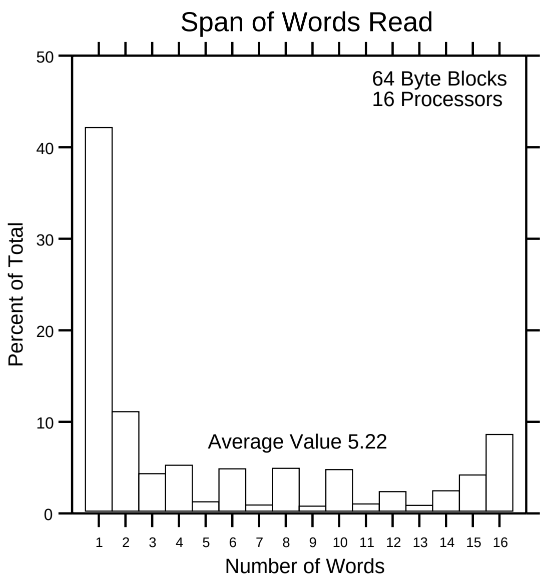


Figure 17: Distance between the furthest spread read words (span) until invalidation, 64-byte blocks.

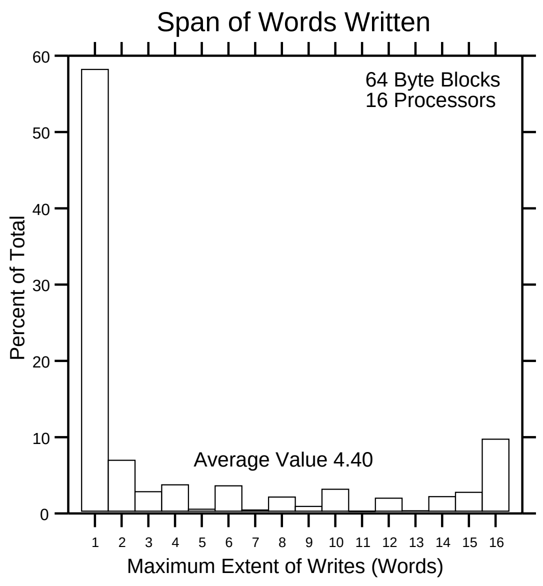


Figure 18: Distance between the furthest spread written words (span) in 64-byte blocks during an invalidation interval.

migratory behavior by overwriting the entire block each time a molecule data structure passes between processors.

Tables 9 and 10 show the average number of distinct words touched and the span of the words touched in each copy of a block during an invalidation interval. On average, the span increases by about 49 percent each time the block size doubles, while the number of distinct words read increases by about 33 percent. Tables 11 and 12 show the average number of distinct words written and the span of the writes between invalidations to the blocks. A stream of writes by one processor can be interrupted by either remote reads or a remote write. Tables 9–12 and Figures 7, 8, 17, and 18 provide insight into the spatial locality of the references to blocks while they reside in the cache. For read references to 64-byte blocks, approximately 42 percent of the blocks use only a single word before being invalidated (Figure 7). Over 60 percent of reads could potentially be satisfied by use of a 16-byte subblock (cumulative sum of values up to 4 words in the histogram in Figure 17). The writes tend to be more tightly concentrated that the reads, with an average increase in the span of 40 percent for each doubling of block size, with the number of distinct words growing by 29 percent. These factors show that subblock fetching for shared blocks, if used judiciously, could tremendously reduce the bus traffic without greatly increasing the number of misses.

For most cases, more distinct words are read during phase 2 of the invalidation interval than are written during phase 1. Occasionally this assumption fails. For example, in **LocusRoute** there is a global data structure which contains a number of read-mostly variables plus some number of values that are updated. The writable variables are protected with locks. When a processor runs out of work to do, it has essentially a spin-loop that checks if it has a new task on its queue. The variable on which it spins is spatially (memory-wise) adjacent to the lock protecting that variable. In addition, this data structure is kept as an array, so possibly several processors are spinning on a single word. Lock variables (lock and unlock operations are classified as writes) that are placed in the same block as variables used for busy-waiting can provide such seeming anomalous behavior as more words are written in a block than are read on average.

Figures 8 and 18 provide some indication how invalidations from sharing can affect a system. An examination of the distinct number of words written (Figure 8) shows that more than half of the time (58%) only a single distinct word is written (possibly multiple times). The companion figure (Figure 18) shows the portion of the block that the writes span (for a 64-byte block). The span of the writes is one word for 58 percent of the cases; about 10 percent of the writes touch words on opposing ends of a block (full span of the block). The average values of the spans are reasonably close to the average number of distinct words words read/written considering that two (or more) distinct words could have a span as large as 16 for a 64-byte block. Because the words touched are likely to be very near each other, subblock protocols would appear to be a good solution for reducing false and dead sharing in these workloads.

Figure 19 shows the fraction of invalidated blocks that

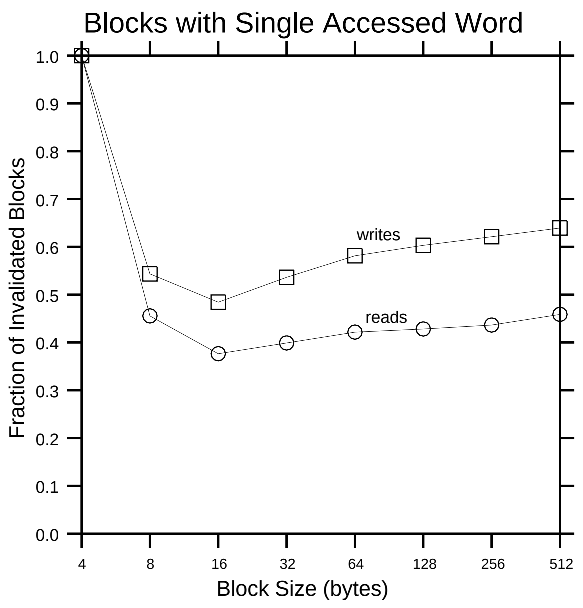


Figure 19: Fraction of invalidated blocks with only a single word read or written.

have only a single word read or written. Both the read and write statistics hit a minimum with 16-byte blocks and increase from there.

E.1 Example Reference Patterns

Table 13 shows example reference patterns to words within shared blocks. These patterns show interesting behavior, which are not necessarily representative of the sharing patterns of the whole program. The accesses are specified by the large number (processor ID) with superscripts for the operation and subscripts for the word accessed within a block (in hexadecimal). The block size is 64 bytes for these samples. The operations on the words can be write (w), read (r), successful lock (l), failed lock (f), and unlock (u).

For example, the reference pattern for **barnes** shows a false sharing problem: words **6** and **7** are exclusively read and word **a** is a lock. Each successful locking operation results in an invalidation of the block from a number of caches, even though the operations on the read-only words are not related. In addition to the useless invalidation of read-shared data, most of the words are not even used, but must be wastefully be transferred between the caches. The ability to perform smaller granularity invalidations would appear to be a large benefit in cases such as this. Other examples of traces showing false and dead sharing behavior in Table 13 are **mp3d**, **pverify**, **topopt**, **water**. **Cholesky** shows an example of multiple processors waiting to acquire a lock. **Ocean** shows a number of processors overwriting the same word without examining the value written by previous writers. **LocusRoute** exhibits a number of processors repeatedly reading a particular word (**b**), with apparently unrelated writes to other words. **Raytrace** provides a good example of dead sharing, where only 2 words are accessed, of which one shows only read references. In such a case, 14 words are wastefully transferred between caches. In addition, the writes also invalidate the read-only word, indicating false sharing as well.

Breakdown of Misses, Normalized to 4–Byte Blocks																	
Workload	Block Size	Total Misses	True Sharing			False Sharing			Workload	Block Size	Total Misses	True Sharing			False Sharing		
			Fetch Misses	Inv. Misses	Cold Start Misses	hit–> fmiss	hit–> imiss	imiss–> fmiss				Fetch Misses	Inv. Misses	Cold Start Misses	hit–> fmiss	hit–> imiss	imiss–> fmiss
barnes	4	1.000	0.717	0.102	0.181	0.000	0.000	0.000	cholesky	4	1.000	0.159	0.376	0.465	0.000	0.000	0.000
	8	0.578	0.404	0.058	0.099	0.012	0.000	0.001		8	0.506	0.081	0.189	0.234	0.003	0.000	0.000
	16	0.411	0.246	0.037	0.053	0.064	0.005	0.002		16	0.270	0.045	0.101	0.121	0.003	0.001	0.000
	32	0.331	0.144	0.024	0.030	0.114	0.012	0.002		32	0.151	0.027	0.056	0.063	0.004	0.001	0.000
	64	0.322	0.093	0.018	0.018	0.160	0.025	0.002		64	0.091	0.018	0.033	0.034	0.005	0.001	0.000
	128	0.326	0.065	0.016	0.010	0.186	0.040	0.006		128	0.062	0.015	0.021	0.018	0.007	0.001	0.000
	256	0.378	0.059	0.016	0.005	0.239	0.047	0.009		256	0.049	0.014	0.015	0.010	0.008	0.001	0.001
	512	0.429	0.059	0.017	0.003	0.288	0.046	0.012		512	0.047	0.015	0.014	0.006	0.010	0.001	0.002
fmm	4	1.000	0.243	0.173	0.584	0.000	0.000	0.000	locus	4	1.000	0.415	0.273	0.312	0.000	0.000	0.000
	8	0.519	0.126	0.088	0.302	0.001	0.000	0.001		8	0.803	0.381	0.222	0.176	0.014	0.007	0.000
	16	0.297	0.075	0.048	0.165	0.003	0.003	0.001		16	0.444	0.194	0.116	0.094	0.023	0.013	0.000
	32	0.184	0.049	0.027	0.094	0.006	0.005	0.001		32	0.289	0.099	0.064	0.050	0.052	0.020	0.000
	64	0.142	0.036	0.017	0.055	0.018	0.014	0.001		64	0.194	0.052	0.037	0.028	0.055	0.020	0.000
	128	0.122	0.029	0.011	0.034	0.026	0.019	0.001		128	0.139	0.030	0.024	0.015	0.051	0.017	0.000
	256	0.122	0.025	0.009	0.022	0.038	0.025	0.001		256	0.120	0.019	0.020	0.009	0.054	0.016	0.001
	512	0.134	0.022	0.007	0.014	0.053	0.035	0.001		512	0.115	0.016	0.021	0.005	0.055	0.015	0.001
mp3d	4	1.000	0.482	0.477	0.041	0.000	0.000	0.000	ocean	4	1.000	0.533	0.394	0.073	0.000	0.000	0.000
	8	0.556	0.260	0.268	0.021	0.005	0.002	0.000		8	0.539	0.301	0.201	0.037	0.000	0.000	0.000
	16	0.341	0.157	0.162	0.011	0.007	0.004	0.001		16	0.549	0.290	0.107	0.021	0.029	0.103	0.000
	32	0.233	0.106	0.109	0.006	0.007	0.004	0.001		32	0.493	0.257	0.063	0.012	0.044	0.117	0.000
	64	0.187	0.083	0.084	0.003	0.008	0.004	0.004		64	0.488	0.241	0.040	0.007	0.073	0.126	0.000
	128	0.175	0.073	0.071	0.002	0.012	0.008	0.009		128	0.496	0.224	0.031	0.005	0.098	0.137	0.000
	256	0.183	0.072	0.069	0.001	0.017	0.010	0.013		256	0.515	0.210	0.027	0.003	0.119	0.156	0.000
	512	0.216	0.075	0.071	0.001	0.031	0.020	0.017		512	0.516	0.193	0.023	0.002	0.139	0.158	0.000
pthor	4	1.000	0.548	0.186	0.267	0.000	0.000	0.000	pverify	4	1.000	0.526	0.329	0.145	0.000	0.000	0.000
	8	0.956	0.478	0.169	0.169	0.089	0.036	0.005		8	2.840	0.519	0.324	0.106	1.023	0.865	0.002
	16	0.839	0.377	0.136	0.103	0.145	0.046	0.008		16	4.195	0.516	0.322	0.079	2.002	1.273	0.003
	32	0.790	0.341	0.124	0.061	0.188	0.044	0.010		32	5.381	0.514	0.320	0.059	3.045	1.438	0.003
	64	0.784	0.326	0.114	0.036	0.223	0.048	0.014		64	6.163	0.511	0.320	0.042	3.775	1.512	0.003
	128	0.803	0.322	0.109	0.022	0.260	0.050	0.016		128	6.382	0.501	0.319	0.029	4.051	1.479	0.003
	256	0.826	0.317	0.102	0.013	0.300	0.050	0.020		256	7.183	0.366	0.318	0.015	4.750	1.730	0.003
	512	0.922	0.316	0.099	0.008	0.399	0.051	0.026		512	7.769	0.187	0.283	0.008	5.249	2.037	0.004
raytrace	4	1.000	0.088	0.076	0.836	0.000	0.000	0.000	topopt	4	1.000	0.755	0.238	0.008	0.000	0.000	0.000
	8	0.586	0.074	0.051	0.456	0.005	0.000	0.000		8	1.490	0.663	0.147	0.005	0.417	0.228	0.029
	16	0.372	0.066	0.040	0.260	0.006	0.000	0.000		16	1.851	0.587	0.103	0.003	0.768	0.341	0.038
	32	0.263	0.063	0.034	0.153	0.011	0.001	0.000		32	2.431	0.515	0.084	0.002	1.301	0.458	0.055
	64	0.196	0.060	0.031	0.088	0.015	0.001	0.000		64	2.987	0.424	0.076	0.002	1.926	0.478	0.060
	128	0.156	0.060	0.029	0.050	0.015	0.001	0.000		128	3.376	0.334	0.074	0.001	2.455	0.427	0.063
	256	0.135	0.060	0.029	0.029	0.014	0.001	0.000		256	3.612	0.225	0.056	0.001	2.900	0.340	0.065
	512	0.124	0.061	0.029	0.017	0.014	0.002	0.000		512	3.806	0.188	0.057	0.000	3.107	0.366	0.057
volrend	4	1.000	0.066	0.027	0.907	0.000	0.000	0.000	water	4	1.000	0.527	0.422	0.052	0.000	0.000	0.000
	8	0.590	0.042	0.017	0.523	0.006	0.002	0.000		8	0.508	0.269	0.211	0.026	0.002	0.000	0.000
	16	0.369	0.028	0.012	0.318	0.008	0.003	0.000		16	0.289	0.153	0.117	0.014	0.006	0.000	0.000
	32	0.241	0.020	0.009	0.199	0.009	0.004	0.000		32	0.188	0.099	0.070	0.008	0.012	0.000	0.000
	64	0.172	0.016	0.008	0.134	0.009	0.004	0.000		64	0.149	0.073	0.047	0.005	0.024	0.000	0.000
	128	0.131	0.014	0.008	0.095	0.010	0.004	0.000		128	0.132	0.058	0.035	0.003	0.036	0.000	0.000
	256	0.096	0.013	0.007	0.060	0.010	0.005	0.000		256	0.132	0.047	0.031	0.002	0.051	0.001	0.000
	512	0.070	0.013	0.007	0.032	0.012	0.005	0.000		512	0.133	0.035	0.030	0.001	0.065	0.002	0.000

Table 6: Breakdown of workload misses into various types of false sharing and true sharing misses for 16 processors, normalized to misses for 4-byte blocks.

Read Overlap with Preceding Writes								
Workload	Block Size (bytes)							
	4	8	16	32	64	128	256	512
barnes	1.000	1.648	1.993	1.905	1.363	0.848	0.449	0.245
cholesky	1.000	0.233	0.219	0.193	0.175	0.139	0.104	0.072
fmm	1.000	1.891	2.963	3.948	3.768	3.572	3.118	2.551
locus	1.000	0.993	1.868	2.578	3.509	4.387	4.091	3.307
mp3d	1.000	1.340	1.619	1.127	0.518	0.167	0.095	0.054
ocean	1.000	1.795	1.675	1.715	1.532	1.306	1.004	0.807
pthor	1.000	0.842	0.783	0.711	0.626	0.542	0.445	0.311
pverify	1.000	0.293	0.103	0.045	0.030	0.021	0.017	0.011
raytrace	1.000	0.965	1.135	0.952	0.824	0.699	0.638	0.617
topopt	1.000	1.016	0.857	0.661	0.453	0.331	0.216	0.120
volrend	1.000	1.739	3.038	5.159	7.791	10.173	11.508	11.252
water	1.000	1.985	3.873	7.513	10.756	11.151	6.060	3.539
Average	1.000	1.228	1.677	2.209	**2.612**	2.778	2.312	1.907

Table 7: Average set of words modified in the write phase of an invalidation interval that are read by other processors during the read phase.

Write Overlap with Preceding Writes								
Workload	Block Size (bytes)							
	4	8	16	32	64	128	256	512
barnes	1.000	1.391	1.478	1.139	0.859	0.420	0.285	0.172
cholesky	1.000	1.968	3.458	5.679	8.028	9.763	10.278	9.021
fmm	1.000	1.843	3.092	4.800	5.828	4.963	4.061	3.336
locus	1.000	1.197	2.056	3.091	4.248	5.249	5.575	4.824
mp3d	1.000	1.750	2.511	2.868	2.505	1.816	1.268	0.724
ocean	1.000	1.152	1.146	1.140	1.118	1.100	1.090	1.065
pthor	1.000	0.785	0.743	0.588	0.489	0.411	0.343	0.263
pverify	1.000	0.225	0.127	0.101	0.088	0.074	0.052	0.038
raytrace	1.000	1.193	1.275	1.321	1.230	1.165	1.027	0.949
topopt	1.000	0.259	0.174	0.115	0.095	0.085	0.074	0.056
volrend	1.000	0.855	0.778	0.722	0.691	0.659	0.612	0.520
water	1.000	1.915	3.123	4.172	4.603	4.728	4.599	4.568
Average	1.000	1.211	1.663	2.145	**2.482**	2.536	2.439	2.128

Table 8: Average set of words written during invalidation interval that overlap words written in a succeeding invalidation interval with different writing processors.

Distinct Reads								
Workload	Block Size (bytes)							
	4	8	16	32	64	128	256	512
barnes	1.000	1.769	2.803	4.372	5.770	7.553	8.760	10.216
cholesky	1.000	1.967	3.575	6.120	9.260	12.355	14.565	14.487
fmm	1.000	1.919	3.181	4.877	6.337	8.344	9.750	11.139
locus	1.000	1.199	2.298	3.761	5.827	8.787	11.166	13.523
mp3d	1.000	1.875	3.077	4.556	5.824	6.910	7.619	7.922
ocean	1.000	1.779	1.894	2.135	2.357	2.548	2.674	2.855
pthor	1.000	1.176	1.581	1.925	2.264	2.738	2.901	2.915
pverify	1.000	1.007	1.028	1.164	1.573	2.047	2.773	3.468
raytrace	1.000	1.229	1.401	1.644	1.915	2.248	2.645	3.076
topopt	1.000	1.096	1.355	1.361	1.593	1.981	2.485	2.780
volrend	1.000	1.563	2.207	2.835	3.306	3.629	3.824	3.969
water	1.000	1.956	3.387	5.015	6.156	7.645	11.219	13.732
Average	1.000	1.545	2.316	3.314	**4.349**	5.565	6.699	7.507

Table 9: Number of distinct words read in a block per processor before the block is invalidated.

Span Reads								
Workload	Block Size (bytes)							
	4	8	16	32	64	128	256	512
barnes	1.000	1.769	2.864	4.740	6.529	10.615	14.319	22.955
cholesky	1.000	1.967	3.621	6.263	9.643	11.830	14.957	16.746
fmm	1.000	1.919	3.201	5.132	7.048	9.951	14.402	22.827
locus	1.000	1.199	3.002	5.459	9.405	15.606	21.650	28.229
mp3d	1.000	1.875	3.146	4.725	6.390	7.135	8.000	8.586
ocean	1.000	1.779	1.895	2.344	2.928	4.063	7.390	12.610
pthor	1.000	1.176	1.784	2.339	3.311	4.173	5.173	7.405
pverify	1.000	1.007	1.029	1.289	2.462	4.869	13.642	33.013
raytrace	1.000	1.229	1.403	1.831	2.719	3.807	5.435	7.499
topopt	1.000	1.096	1.418	1.595	2.192	2.861	6.934	10.254
volrend	1.000	1.563	2.268	3.004	3.722	4.178	4.729	5.059
water	1.000	1.956	3.391	5.026	6.245	7.782	11.947	17.504
Average	1.000	1.545	2.418	3.646	**5.216**	7.239	10.715	16.057

Table 10: Span of words read in a block between invalidations.

Distinct Writes								
Workload	Block Size (bytes)							
	4	8	16	32	64	128	256	512
barnes	1.000	1.610	2.201	2.760	3.538	4.193	3.539	2.961
cholesky	1.000	1.982	3.661	6.421	10.074	13.865	16.710	16.988
fmm	1.000	1.908	3.401	5.650	8.092	10.111	10.508	10.913
locus	1.000	1.318	2.484	4.214	6.648	9.856	13.567	16.821
mp3d	1.000	1.780	2.940	4.369	5.496	6.158	6.584	6.588
ocean	1.000	1.184	1.187	1.201	1.277	1.575	2.151	3.654
pthor	1.000	1.096	1.335	1.462	1.526	1.571	1.599	1.606
pverify	1.000	1.002	1.033	1.082	1.175	1.422	1.487	1.443
raytrace	1.000	1.338	1.591	1.797	1.926	2.000	2.036	2.043
topopt	1.000	1.020	1.332	1.166	1.092	1.061	1.118	1.171
volrend	1.000	1.287	1.451	1.585	1.740	1.952	2.077	2.186
water	1.000	1.947	3.250	4.495	5.145	5.462	5.670	6.089
Average	1.000	1.456	2.156	3.017	**3.977**	4.936	5.587	6.039

Table 11: Number of distinct words written to a block before a read by another processor.

Span Writes								
Workload	Block Size (bytes)							
	4	8	16	32	64	128	256	512
barnes	1.000	1.610	2.217	2.855	4.086	6.192	5.411	4.858
cholesky	1.000	1.982	3.662	6.480	10.330	14.866	19.468	22.363
fmm	1.000	1.908	3.417	5.713	8.302	10.485	12.043	16.057
locus	1.000	1.318	3.003	5.488	9.552	16.229	25.900	36.005
mp3d	1.000	1.780	3.078	4.662	6.178	7.445	8.686	9.565
ocean	1.000	1.184	1.187	1.201	1.277	1.596	2.226	4.165
pthor	1.000	1.096	1.386	1.650	1.883	2.194	3.086	4.694
pverify	1.000	1.002	1.033	1.084	1.178	2.307	3.720	6.750
raytrace	1.000	1.338	1.598	1.819	1.956	2.107	2.208	2.361
topopt	1.000	1.020	1.334	1.220	1.098	1.099	2.746	3.805
volrend	1.000	1.287	1.452	1.588	1.757	1.982	2.329	2.614
water	1.000	1.947	3.255	4.508	5.257	5.924	7.327	11.890
Average	1.000	1.456	2.219	3.189	**4.405**	6.035	7.929	10.427

Table 12: Span of words written to a block before a read by another processor.

Sample Multiprocessor Reference Patterns	
barnes	$13^l_a 13^u_a 14^r_7 14^r_6 7^r_7 7^r_6 4^r_7 4^r_6 0^r_7 0^r_6 9^r_7 9^r_6 11^r_7 11^r_6 1^r_7 1^r_6 6^r_7 6^r_6 10^l_a$ $10^u_a 5^l_a 5^u_a 15^l_a 0^r_7 0^r_6 15^u_a 4^r_7 4^r_6 2^l_a 2^u_a 7^l_a 8^f_a 8^f_a 8^f_a 8^f_a 8^f_a 8^f_a 8^f_a$
cholesky	$2^l_1 11^w_6 15^w_3 0^w_0 11^l_b 10^w_2 0^l_4 5^f_1 5^f_1 5^f_1 5^f_1 14^f_4 5^f_1 14^f_4 5^f_1 14^f_4 5^f_1 14^f_4 5^f_1$ $14^f_4 5^f_1 14^f_4 5^f_1 14^f_4 5^f_1 14^f_4 5^f_1 14^f_4 5^f_1 14^f_4 5^f_1 14^f_4 5^f_1 14^f_4 2^u_1 5^l_1 14^f_4 14^f_4$
locus	$3^w_6 3^r_6 14^r_b 9^r_b 1^r_b 11^r_b 13^r_b 5^r_b 10^r_b 4^r_b 8^r_b 15^r_b 14^r_b 9^r_b 1^r_b 11^r_b 13^r_b 5^r_b 10^r_b 4^r_b 8^r_b 15^r_b$ $14^r_b 9^r_b 1^r_b 3^r_9 11^r_b 13^r_b 5^r_b 10^r_b 4^r_b 8^r_b 15^r_b 3^w_9 3^r_9 14^r_b 3^r_1 9^r_b 1^r_b 11^r_b 13^r_b 5^r_b 10^r_b 4^r_b 8^r_b$
mp3d	$7^l_c 7^r_d 7^r_e 7^r_f 7^w_e 7^w_f 7^w_d 7^u_c 10^l_b 10^r_5 10^r_6 10^r_8 10^w_5 10^r_7 10^w_6 10^r_a 10^w_7 10^r_9 10^w_8$ $10^w_a 10^w_9 10^u_b\ 10^l_0 10^r_1 10^r_2 10^r_3 10^w_2 10^w_3 10^w_1 10^r_4 10^u_0 13^l_b 13^r_5 13^r_6 13^r_8 13^w_5$
ocean	$12^r_a 4^r_a 8^r_a 12^r_b 4^r_b 8^r_b 0^w_2 0^w_3 11^r_a 13^r_a 14^r_a 15^r_a 5^r_a 6^r_a 7^r_a 9^r_a 10^r_a 11^r_b 13^r_b$ $14^r_b 15^r_b 5^r_b 6^r_b 7^r_b 9^r_b 10^r_b 12^w_2 4^w_2 8^w_2 12^w_3 1^w_2 2^w_2 3^w_2 4^w_3 8^w_3 1^w_3 2^w_3 3^w_3$
pverify	$7^r_4 7^w_4 12^r_9 12^w_9 7^r_4 12^r_9 4^r_1 4^w_1 4^r_1 9^r_6 7^r_4 13^r_a 13^w_a 13^r_a 3^r_0 3^w_0 6^r_3 10^r_7$ $6^w_3 3^r_0 6^r_3 14^r_b 7^r_4 7^w_4 7^r_4 15^r_c 15^w_c 9^r_6 8^r_5 15^r_c 8^w_5 8^r_5 10^r_7 9^r_6 9^w_6 9^r_6$
raytrace	$14^r_2 14^w_2 5^r_2 5^w_2 3^r_2 3^w_2 11^r_2 11^w_2 11^r_4 3^r_4 14^r_2 14^w_2 5^r_2 5^w_2 14^r_4 12^r_2 12^w_2 12^r_4$ $5^r_4 7^r_2 7^w_2 7^r_4 2^r_2 2^w_2 2^r_4 10^r_2 10^w_2 4^r_2 4^w_2 10^r_2 10^w_2 10^r_4 14^r_2 14^w_2 14^r_4 4^r_2 4^w_2$
topopt	$11^w_8 13^r_e 9^w_2 13^w_e 9^r_3 9^w_3 11^r_7 11^w_7 13^r_d 13^w_d 8^r_0 9^r_1 8^w_0 9^w_1 11^r_8 11^w_8 13^r_e 13^w_e$ $9^r_2 9^w_2 11^r_9 11^w_9 13^r_f 13^w_f 9^r_3 9^w_3 8^r_0 8^w_0 11^r_7 11^w_7 13^r_d 13^w_d 11^r_8 11^w_8 13^r_e 13^w_e$
water	$4^r_b 4^r_a 4^r_f 4^r_e 4^r_d 4^r_c 4^r_d 4^r_c 12^r_0 12^r_1 12^w_0 12^w_1 12^r_2 12^r_3 12^w_2 12^w_3 12^r_4 12^r_5 12^w_4$ $12^w_5 12^r_6 12^r_7 12^w_6 12^w_7 12^r_8 12^r_9 12^w_8 12^w_9 12^r_a 12^r_b 12^w_a 12^w_b 12^r_c 12^r_d 12^w_c 12^w_d$

Table 13: Sample shared memory block reference sequences for some workloads. The large number corresponds to the processor ID, subscripts are the word accessed (in hexadecimal), the superscript is the operation to the word (**w**: write, **r**: read: **l**: successful lock, **u**: unlock, **f**: failed lock attempt).