The Detection and Elimination of Useless Misses in Multiprocessors

Michel Dubois, Jonas Skeppstedt*, Livio Ricciulli,
Krishnan Ramamurthy, and Per Stenström*

Dept. of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562, U.S.A.

*Department of Computer Engineering
Lund University
P.O. Box 118, S-221 00 LUND, Sweden

Abstract

In this paper we introduce a new classification of misses in shared-memory multiprocessors based on interprocessor communication. We identify the set of essential misses, i.e., the smallest set of misses necessary for correct execution. Essential misses include cold misses and true sharing misses. All other misses are useless misses and can be ignored without affecting the correctness of program execution. Based on the new classification we compare the effectiveness of five different protocols which delay and combine invalidations leading to useless misses. In cache-based systems the protocols are very effective and have miss rates close to the essential miss rate. In virtual shared memory systems the techniques are also effective but leave room for improvements.

1.0 Introduction

The design of efficient memory hierarchies for shared-memory multiprocessors is an important problem in computer architecture today. With current interconnection and memory technologies, the shared-memory access time is usually too large to maintain good processor efficiency. As the number and the speed of processors increase, it becomes critical to keep instructions and data close to each processor.

These considerations have led to two types of systems, cache-based systems [5,15] and virtual shared memory systems [1,2], in which multiple copies of the same data may exist and coherence must be maintained among them. Cache-based systems are shared memory systems in which each processor has a private cache with a block size typically less than 256 bytes. A virtual shared memory system is a shared memory system built on top of a distributed (message-passing) multicomputer, through software management of virtual memory; coherence is maintained on pages of size larger than 512 bytes.

Write invalidate protocols are the most widely used protocols to enforce consistency among the copies of a particular block or page. In such protocols, multiple processors may have a copy of a block or page provided no one modifies it, i.e., the copy is *shared*. When a processor needs to write into a copy, it must first acquire *ownership*, i.e., it must have the sole copy among all caches. Acquiring ownership implies that copies in remote caches must be invalidated. While an invalidation or miss request is pending in a processor, the processor must often be blocked. The processor blocking time during a memory request is called the *penalty* of the request. Whereas invalidation penalties can be easily eliminated through more aggressive consistency models [9,13], load miss latencies are much harder to hide.

In write invalidate protocols, misses can be classified in cold, replacement, and coherence misses. A cold miss occurs at the first reference to a given block by a given processor. Subsequent misses to the block by the processor are either caused by invalidations (coherence misses) or by replacements (replacement misses). A large number of the coherence misses are not needed for the correct execution of a parallel program. Intuitively, only coherence misses which communicate new values to a processor are essential; all other coherence misses are useless.

Our first contribution is to identify the sets of essential and useless misses in an execution. This new classification is important in general because we cannot compare approaches to reduce miss rates without good measures of miss rate components. For example, in compiler-based approaches to miss reduction it is important to understand how much improvement is due to the elimination of useless misses and how much is due to better locality. Moreover, by identifying the minimum miss rate for an execution, we can understand how close we are to the minimum miss rate and whether further improvements are possible.

This research was supported by the National Science Foundation under Grant No. CCR-9115725

0884-7495/93 $3.00 © 1993 IEEE

Our second contribution consists of applying the classification to evaluate effects of invalidation timing — or *scheduling* — on the miss rate and its components. Aggressive techniques to tolerate memory latency tend to alter the timing of invalidations. For example, in the DASH machine [13], stores are issued by the processor immediately in a store buffer and are executed later on in the cache and in the system. Therefore invalidations are delayed both in the local processor and later on when they are propagated in the system. These delays can cut the number of useless misses. We show an invalidation schedule which yields the minimum possible miss rate for a trace (called the *essential* miss rate). In practice, it is difficult to reach that minimum but we compare the effectiveness of attempts to do so. We also present results for a worst-case propagation of invalidations consistent with release consistency [13] and which, in some cases, causes a large number of misses.

The paper is structured as follows. Section 2 covers the classification and is followed by the description of various protocols to eliminate useless misses and of the worst-case schedule of invalidations in section 3. In section 4, we present and justify the experimental methodology. Finally, in sections 5, 6 and 7 we analyze our simulation results.

2.0 Classification of Misses

In contrast with previous attempts to classify coherence misses in multiprocessors [11,16] our classification is based on interprocessor communication. We will restrict ourselves to infinite caches and write invalidate protocols. We define the ***lifetime*** of a block in the cache following a miss as the time interval during which the block remains valid in the cache (from the occurrence of the miss until its invalidation or until the end of the simulation).We then define the following classes of misses.

Cold miss: The first miss to a given block by a processor.

Essential miss: The first miss to a given block by one processor (cold miss) is an essential miss. A miss is also an essential miss if, during the *lifetime* of the block in the cache, the processor *accesses* (Important note: an *access* can be a load or a store[1]) a value *defined* by a different processor since the last essential miss experienced by the same processor to the same block.

Pure True Sharing miss (PTS): An essential miss which is not cold.

Pure False Sharing miss (PFS): A miss which is not essential (useless miss).

The first miss in a processor for a given block is a cold miss. This miss communicates all the initial values plus all the values modified by other processors since the start of the simulation. Assume that, after this cold miss, other processors change the value of word(s) in the block. The first true sharing miss is detected when the processor *accesses* one of the new values during the lifetime of the block following the miss; at the occurrence of this first true sharing miss all values modified in the block since the cold miss are also communicated to the processor. Between the cold miss and the first true sharing miss there may have been several false sharing misses. These intervening false sharing misses are useless in the sense that the execution from the cache would still be correct if the block had not been loaded and the processor had kept accessing the values loaded on the cold miss instead. True sharing misses are detected one after the other by detecting the first access to a value modified since the previous essential miss.

In some cases it may be useful to refine the definition of cold misses as follows.

Pure Cold miss (PC): Cold miss on a block which has not been modified since the start of the simulation.

Cold and True Sharing miss (CTS): Cold miss on a block modified since the start of the simulation; moreover, during the subsequent lifetime of the block in the cache, the processor *accesses* one of the modified values. Fig. 1 and 2 show examples of CTS misses.

Cold and False Sharing miss (CFS): Cold miss on a block modified since the start of the simulation; however, during the lifetime of the block in the cache, the processor does not access any one of these modified values. Fig. 3 shows an example of a CFS miss.

PC misses can be eliminated by preloading blocks in the cache. CFS misses can be eliminated by preloading blocks in the cache if we also have a technique to detect and eliminate false sharing misses. CTS misses cannot be eliminated. The classification algorithm for PTS, PFS and cold misses (PC+CTS+CFS) is shown in Appendix A.

2.1 Effect of Block Size

The number of essential misses observed in a trace cannot increase when the block size increases, for the following reasons. Cache block sizes increase in power of two. If a referenced word is in a block of size B_1 and in a block of size B_2 such that B_1 is smaller than B_2, then B_1 is included in B_2. The *i*th essential miss for a system with block size B_1 cannot happen after the *i*th essential miss for a system with block size B_2, because each miss in a system with block size B_2 brings more values into the cache than

1. Strictly speaking, the value of the data is not needed for the correct execution of a store. However, shared memory systems require that the value is accessed for the store to complete.

in a system with block size B_1. Similarly the total number of cold misses cannot increase with the block size because more values are brought in on each miss in systems with larger block sizes.

The number of PTS misses decreases with the block size in general, but this is by no means certain. Consider the sequence in Fig. 1, where `Load i` and `Store i` correspond to a load and store in word i and different lines correspond to successive references in the trace. When the block size goes from one word to two words, the number of essential misses decreases, the number of cold misses decreases, and the number of PTS misses increases. In general, when the block size increases, some CTS misses turn into PTS misses; however, the total number of CTS plus PTS misses cannot grow, for the same reason as for the essential misses, i.e., more values are communicated at each essential miss.

Figure 1: Effect of the block size on the number of PTS misses. Words 0 and 1 are in the same block of size 2.

Ref.	P1	P2	B= 1 word	B= 2 words
T0:	Store 0		PC	PC
T1:		Load 0	CTS	CTS
T2:	Store 1	INV	PC	-
T3:		Load 1	CTS	PTS

2.2 Detection and Elimination of Useless Misses

In this section we introduce a simple write-through protocol (with allocation on write misses), which totally eliminates useless misses. On a store into the block, the address of the modified word is propagated to all processors with a copy of the block and is buffered in an invalidation buffer; a local access to a word whose address is present in the buffer invalidates the block copy and triggers a PTS miss. (The invalidation buffer could be implemented by a dirty bit associated with each word in each block of the cache.) This implementation "mimics" the essential miss detection algorithm and its miss rate is the essential miss rate of the trace.

Write-through caches generate an unacceptable amount of write traffic. To make the protocol write back we need to maintain ownership. Stores accessing non-owned blocks with a pending invalidation for ANY one of its words in the local invalidation buffer must trigger a miss. These additional misses are the cost of maintaining ownership.

2.3 Invalidation Delaying and Combining

In the above protocols, invalidations are *delayed* and *combined* in the invalidation buffer until an invalidation leading to an essential miss is detected. Invalidation combining can be done at both ends: before it is sent out and after it is received. Delaying the sending of a store may increase the false sharing miss rate when, for example, the store is delayed across an essential miss in the receiving processor (without the delay, it would have been combined with that essential miss, but after the delay it may create a new miss.) Actually, it may even increase the essential miss rate as shown in Fig. 2. By delaying the second store of P1 from T1 to T2, a new PTS miss is created. Delaying stores at the sending end can only help if the delays lead to the combining of invalidations. Combining invalidations at the receiving end is more effective because the references causing essential misses can easily be detected and combined invalidations originate from all processors.

Figure 2: Effect of trace interleaving on the number of essential misses. Words 0 and 1 are in the same block.

Ref.	P1	P2	Class.
T0:	Store 0		PC
T1:		Load 0	CTS
T2:	Store 1	INV	-
T3:		Load 1	PTS

Ref.	P1	P2	Class.
T0:	Store 0		PC
T1:	Store 1		-
T2:		Load 0	CTS
T3:		Load 1	-

Finally, the essential miss rate is not an intrinsic property of an application, but only a property of an execution (or of an interleaved trace). For example, the two sequences in Fig. 2 are possible and equivalent executions but the second one yields less essential misses.

3.0 Comparison with Previous Classifications

Previous attempts to classify coherence misses, by Eggers and Jeremiassen [11] and by Torrellas, Lam and Hennessy [16][2] did not capture the set of essential misses.We now contrast these two schemes with ours.

3.1 Torrellas' Scheme

In this scheme, a cold miss (CM) is detected if the accessed *word* is referenced for the first time by a given processor. A True Sharing Miss (TSM) is detected on a reference which misses in the cache, accesses a word accessed before, and misses in a system with a block size

2. In the rest of the paper we will refer to these two proposed classifications as "Eggers" and "Torrellas", whereas we will refer to our classification as "ours".

of one. All other misses are False Sharing Misses (FSM).

This approach has several drawbacks. The way cold misses are detected is not conventional. As it is, the classification is only applicable to iterative algorithms in which words are accessed more than once. Many important parallel algorithms, such as FFT and matrix multiply, do not belong to this class.

Figure 3: Basic shortcoming of current schemes. (1):Torrellas. (2):Eggers. (3):Ours.

Ref.	P1	P2	(1)	(2)	(3)
T0:	Store 1		CM	CM	PC
T1:		Load 0	CM	CM	CFS
T2:	Load 1		-	-	-
T3:	Load 0		-	-	-
T4:	INV	Store 0	-	-	-
T5:	Load 1		FSM	FSM	PTS
T6:	Load 0		-	-	-

The major drawback of the classification however is that it depends on which word of the block is accessed first on a miss. Consider the sequence in Fig. 3. The miss at reference T5 brings a new value defined at T4 and accessed at T6 in the cache of processor P1, and yet it is classified as a false sharing miss. If we did not execute the miss at T5 (or equivalently ignored the invalidation at T4) and kept the old block in the cache instead, P1 would read a stale value at T6. In their paper, Torrellas et al. introduce the notion of *prefetching effects*; however they do not attempt to quantify these effects.

3.2 Eggers' Scheme

A cold miss (CM) occurs at the first reference to a given block by a given processor and all following misses to the same block by the same processor are classified as invalidation misses. Invalidation misses are then classified as True Sharing Misses (TSM) if the word accessed on the miss has been modified since (and including) the reference causing the invalidation. All other invalidation misses are classified as False Sharing Misses (FSM).

Figure 4: Differences between Eggers' and Torrellas' classifications. (1):Torrellas. (2):Eggers. (3):Ours.

Ref.	P1	P2	(1)	(2)	(3)
T0:	Load 1		CM	CM	PC
T1:		Load 0	CM	CM	PC
T2:	INV	Store 1	-	-	-
T3:	Load 0		CM	FSM	PFS
T4:	INV	Store 0	-	-	-
T5:	Load 1		TSM	FSM	PTS
T6:	Load 0		-	-	-

Clearly, any true sharing miss in Eggers' classification must also be a true sharing miss in Torrellas'. Fig. 4 shows a sequence such that more true sharing misses are counted by Torrellas' method and illustrates the differences in counting cold misses.

3.3 Comparison Between the Three Classifications

We have run a few traces to see whether there was a significant difference between the different classifications for real data. Table 1 shows some results for some benchmark runs with the larger data set sizes (see section 5), namely LU200 and MP3D10000 for block sizes of 32 bytes and 1,024 bytes. As can be seen from Table 1, current measures of false and true sharing are unreliable. Eggers' scheme exaggerates the amount of false sharing and underestimates true sharing, because it ignores the possibility of communicating new values in references following the miss.

Table 1: Comparison between the classifications

BENCH.	LU	LU	MP3D	MP3D
Block(bytes)	32	1024	32	1024
PTS-ours	5,769	7,941	188,120	82,125
TSM-Eggers	2,845	2,558	178,206	67,447
TSM-Torrellas	597	183	177,272	112,562
COLD-ours	110,955	5,545	46,242	4,058
COLD-Eggers	110,955	5,545	46,242	4,058
COLD-Torrellas	113,812	9,827	52,264	26,011
PFS-ours	11,839	79,882	31,206	266,245
FSM-Eggers	14,763	85,265	41,120	280,923
FSM-Torrellas	14,154	83,358	36,032	213,855

In Torrellas' scheme, this effect is partially compensated by the fact that a new value can cause a true sharing miss even if it has been loaded in a cache on a previous true sharing miss. Torrellas' classification also classifies a large number of true and false sharing misses as cold misses. The numbers in Table 1 indicate that the net effect tends to be an overestimation of the number of false sharing misses as well.

In sections 2.2 and 2.3 approaches to reduce the miss rate were derived from the classification. In general, the times at which invalidations are scheduled can affect the miss rate and the components of the miss rate, as we argued in the introduction. In the rest of the paper, we present simulation results to quantify these effects.

4.0 Scheduling of Invalidations

We have simulated the following schedules of invalidations.

MIN (Write-through with Word Invalidation): This is the write-through protocol of section 2.2 using a dirty bit per word. It has no false sharing and yields the essential miss rate of the trace.

OTF (On-The-Fly Protocol): Each reference is scheduled one by one in the simulation. The miss rate of the OTF protocol is the miss rate usually derived when using trace-driven simulations.

RD (Receive Delayed Protocol): Invalidations are sent without delay and stored in an invalidation buffer when they are received. When a processor executes an *acquire* all blocks for which there is a pending received invalidation are invalidated [8].

SD (Send Delayed Protocol): If the processor is the owner at the time of a store, the store is completed without delay. Otherwise, the store is buffered. Pending stores in the buffer are sent at the execution of a *release*. A received invalidation is immediately executed in the cache [8].

SRD (Send and Receive Delayed Protocol): This protocol combines the features of RD and SD above [8].

WBWI (Write-back with Word Invalidate Protocol): This protocol is a special case of protocols with partial block invalidations [6]. It is similar to the MIN algorithm, but it maintains ownership in order to reduce the write traffic, as described in section 2.2. WBWI is also similar to RD except that it relies on a dirty bit per word to schedule invalidations instead of relying on releases and acquires.

MAX (Worst-case Invalidation Propagation): MAX is not a protocol. Rather, it corresponds to a worst-case scenario for scheduling invalidations, consistent with the release consistency model. Stores from a given processor can be performed at any time between the time they are issued by the processor and the next release in that processor and they can be performed out of program order. Within these limits, we schedule the invalidations of each store so as to maximize the miss rate [10].

In the next section, we describe and justify the experimental methodology used to compare the effects of various schedules of invalidations.

5.0 Simulation Methodology and Benchmarks

Early on in this project we used execution-driven simulation. We quickly ran into problems because modifying the schedule of invalidations resulted in different executions of the benchmarks. Benchmarks would yield different traces due to different scheduling of threads or would even yield different results. The effects of different scheduling of invalidations were buried into the effects of altered executions in unpredictable ways. Therefore, we decided to use trace-driven simulation instead and collected traces for four benchmark programs and two different data set sizes. All benchmarks were run for 16 processors and infinite caches.

In order to evaluate the effects of delayed protocols, applications must be free of data races and conform to the release consistency model. The first three benchmarks are parallel applications developed at Stanford University (MP3D, WATER, and LU) of which the first two are also in the SPLASH suite [14]. These applications are written in C using the Argonne National Laboratory macro package [3] and are compiled with the `gcc` compiler (version 2.0) using optimization level -O2. Traces from these benchmarks were captured by the CacheMire Test Bench, a tracing and simulation tool for shared-memory multiprocessors [4].

MP3D is a 3-dimensional particle simulator used by aerospace researchers to study the pressure and temperature profiles created as an object flies at hypersonic speeds through the upper atmosphere. The overall computation consists of calculating the positions and velocities of particles during a number of time steps. In each iteration (a time step) each processor updates the positions and velocities of each of its particles. When a collision occurs, the processor updates the attributes of the particle colliding with its own. We have run MP3D with 1,000 particles for 20 time steps (referred to as MP3D1000) and with 10,000 for 10 time steps (referred to as MP3D10000). In both cases, the locking option was switched on, to eliminate data races.

WATER performs an N-body molecular dynamics simulation of the forces and potentials in a system of water molecules in the liquid state. The overall computation consists of calculating the interaction of the atoms within each molecule and of the molecules with each other during a number of time steps. As in MP3D, each processor updates its objects in each iteration (time step). Interactions of its molecules with other molecules involve modifying the data structures of the other molecules. We have run WATER with 16 molecules for 10 time steps (WATER16) and with 288 molecules for 4 time steps (WATER288).

LU performs the LU-decomposition of a dense matrix. The overall computation consists of modifying each column based on the values in all columns to its left. Columns are modified from left to right. They are statically assigned to processors in a finely interleaved fashion. Each processor waits until a column has been produced and then

uses it to modify all its columns. We have run LU with a 32x32 (LU32) and a 200x200 random matrix (LU200).

Finally, **JACOBI** [12] was written by us using the ANL macros [3] provided with the SPLASH benchmark suite. It is an iterative algorithm for solving partial differential equations. Two 64x64 grid arrays of double precision floating point numbers (8 bytes each) are modified in turn in each iteration. A component in one grid is updated by taking the average of the four neighbors of the same component in the other grid. After each iteration, the processors synchronize through a barrier synchronization, a test for convergence is done and the two arrays are switched. In each iteration, one array is read only and the other one is write only but across consecutive iterations all components are accessed read/write. Each of the 16 processors is assigned to the update of a 16x16 subgrid.

Table 2: Characteristics of the benchmarks.

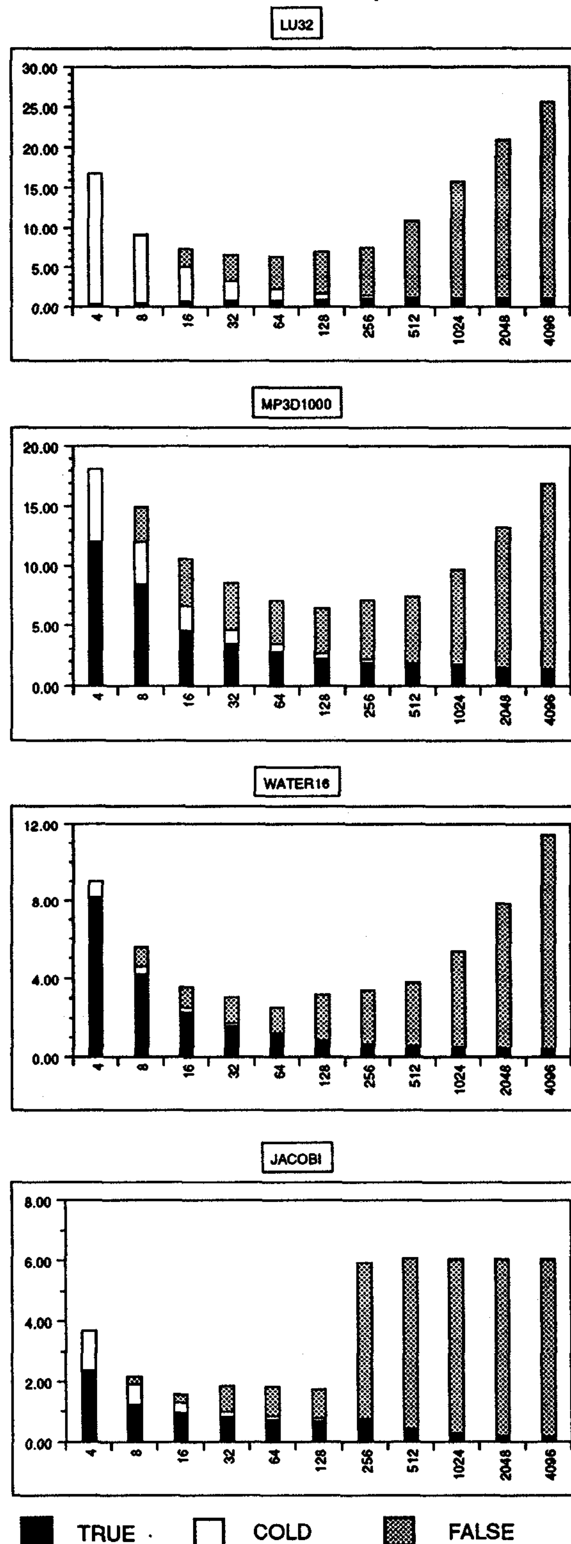
BENCH-MARK	SPEED UP	WRITES (000's)	READS (000's)	ACQ/ REL (000's)	DATA SET (KB)
MP3D1000	10.9	357	948	90	36
MP3D10000	14.9	1,510	2,561	411	360
WATER16	12.3	83	973	9	10
ATER288	14.9	5,114	71,134	531	195
LU32	5.7	37	136	4	8
LU200	14.9	5,663	11,764	10	320
JACOBI	15	280	2,407	4	65

In Table 2, we show the characteristics of the benchmarks in the parallel section. The speedup derivation assumes a perfect memory system (single-cycle latencies). We first show the classification for each benchmark with the smaller problem sizes.

6.0 Miss Classification

In LU32, each column goes through two phases. In the first phase, it is accessed exclusively by a single processor whereas, in the second phase, it is read by many. As a result, the column distribution causes CTS misses which show up for small block sizes in Fig. 5. This component drops until the block size reaches 512 bytes because the largest columns occupy 512 bytes each. As the block size increases the CTS misses turn into PTS misses, an effect identified in Fig. 1. As for false sharing, LU works on triangular matrices and columns are interleaved among processors. False sharing starts to appear for the smaller columns and is significant even for small block sizes.

Figure 5: Miss rate classification for the four benchmarks with small data sets.(Miss rates are in %.)



In MP3D1000, two data structures contribute to the coherence miss rate: the particle and the space-cell

structures. Particle objects occupy 36 bytes each and are finely interleaved among processors. Space cell objects occupy 48 bytes each. In each iteration, all particles are moved. Moving a particle means that its position and the corresponding cell data structures are updated. Occasionally, a particle collides with another particle. This causes true sharing misses if the two colliding particles are allocated to different processors. During a collision five words (20 bytes) of the data structures of the two particles are updated. Consequently, the true sharing miss rate component decreases dramatically up to 32 bytes. False sharing misses are due to modifications of particles and of space cells. False sharing starts to appear for a block size of eight bytes because the object size is 36 bytes and consecutive particle objects belong to different processors. Additional false sharing due to the space-cells appears for blocks larger than 16 bytes because the space-cell object size is 48 bytes.

In WATER16, each processor calculates both intra-molecular and inter-molecular interactions. True sharing is caused by inter-molecular interactions. During this calculation, a part of the other molecule's data structure, corresponding to nine double words (72 bytes), is modified. Consequently, the true sharing miss component decreases rapidly up until a block size of 128 bytes. False sharing mainly results from accesses to two consecutively allocated molecules belonging to different processors. The false sharing rate starts to grow significantly when the block size approaches the size of the molecule data structure (680 bytes).

In JACOBI, each matrix element is a double word (8 bytes) and we would expect true sharing to go down abruptly to half as we move from a block size of 4 to 8 bytes. After that point, it should decrease more slowly. We see these effects in Fig. 5. False sharing appears when a block partly covers the data partitions of two processors. Since the size of a row in the submatrix is 16 elements (128 bytes), false sharing abruptly goes up for a block size of 256 bytes as can be seen in Fig. 5. False sharing starts to appear for a block size of 8 bytes because of the large number of barrier synchronizations in the program (one after each iteration) and also because of the particular implementation of barriers in the ANL macros. In this implementation, two words (a counter and a flag) are stored in consecutive memory locations. The same effect also explains parts of the false sharing present inWATER16 and MP3D1000 for a block size of 8 bytes.

7.0 Effects of Invalidation Scheduling

In Fig. 6 we show the comparison between the miss rates of LU32, MP3D1000, WATER16 and JACOBI for block sizes of 64 (Fig. 6a) and 1,024 bytes (Fig. 6b). These block sizes correspond to cache-based systems and virtual shared memory systems, respectively. The decomposition into PTS, COLD and PFS misses is shown except for MIN (which has no false sharing), WBWI and MAX, for which we only display the total miss rate.

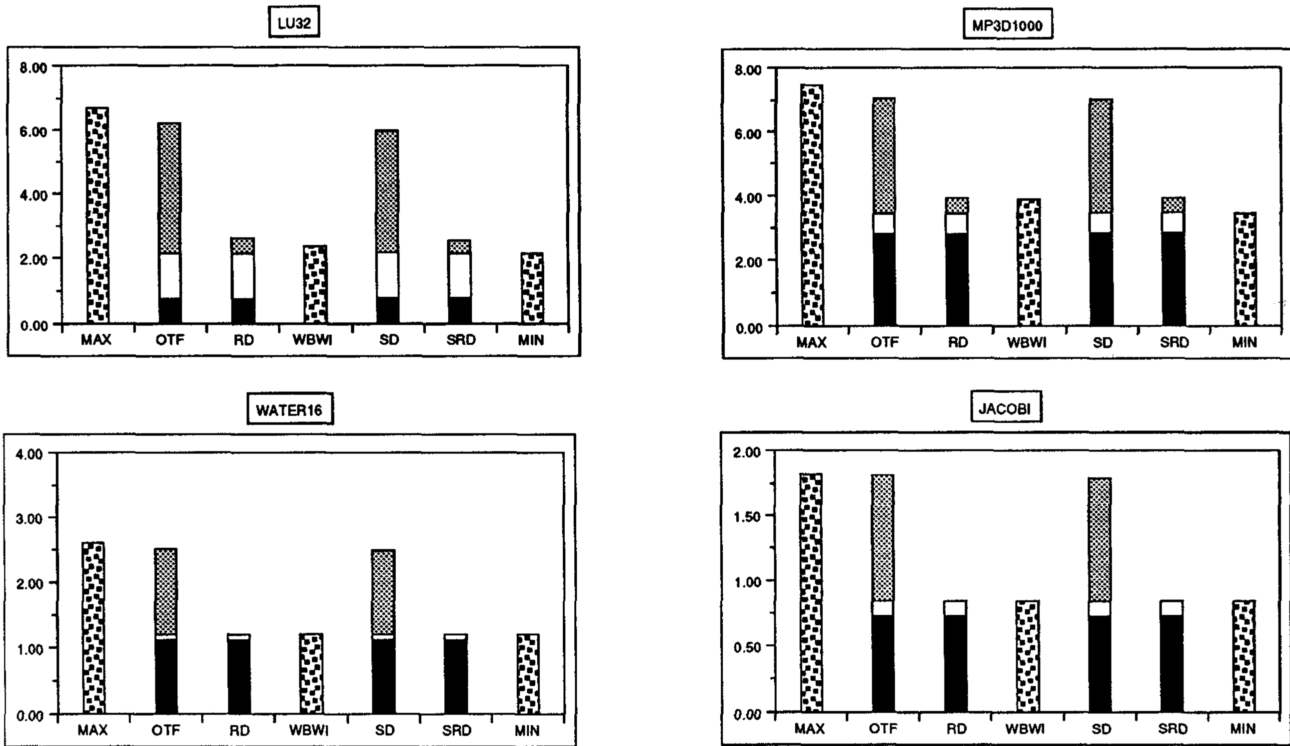
All protocols except MIN suffer from the cost of maintaining ownership. The importance of this effect can be understood by comparing WBWI and MIN, because the only difference between these protocols is ownership. Remarkably, Fig. 6a shows that the cost of ownership is very low for B=64 bytes. This can be explained easily: For small block sizes, the probability that several processors are writing into different parts of the same block at the same time is very low. By contrast the plots for B=1,024 bytes (Fig. 6b) show a large difference between the miss rates of WBWI (or RD) and MIN.

The differences between the essential miss rates of OTF (equal to the miss rate of MIN), RD, SD and SRD are negligible.This shows that the delays do not affect the essential miss count. Store combining at the sending end occurs seldom for B=64 because the blocks are too small. In general, our simulations show that pure send delayed protocols are not very effective for caches and that their miss rate is still far from the essential miss rate. There are much more opportunities for store combining in systems with B=1,024 and the effectiveness of pure SD protocols is much better.

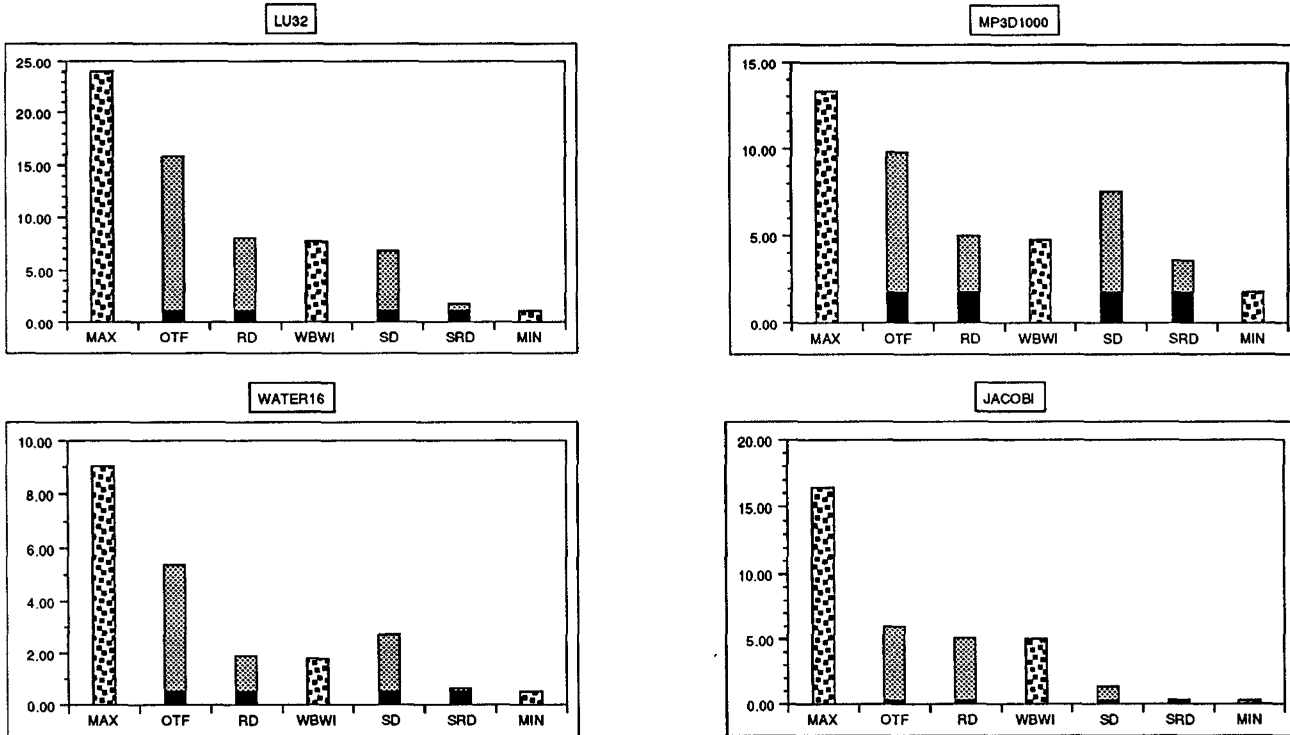
For RD protocols, we expect more false sharing than for the WBWI protocol because the synchronization access causing the invalidation may be very far from the access to the stale word and may even not protect the stale word. For B=64 and B=1,024 it appears from the simulations that this effect is negligible. The invalidation rate of RD is also expected to be lower than that of WBWI because invalidations target entire blocks instead of words. Moreover, WBWI requires one dirty bit per word whereas RD only needs 1 stale bit per block [6,8]. From these results it appears that RD is preferable to WBWI provided that relaxed consistency models are acceptable (WBWI is valid for systems with any consistency model.)

Overall, for B=64, the miss rates of the protocols (except for OTF and SD) are very close to the essential miss rate of the trace and therefore there is little room for improvements. We want to stress here the importance of a correct classification. For example, for LU32 and for B=64 bytes, Eggers' classification yields an essential miss rate (CM+TSM) of 1.68%, whereas our classification yields an essential miss rate of 2.14%. The miss rate of WBWI is 2.37% and therefore is very close to the minimum possible. Eggers' classification would have led us to believe that significant additional reductions of the miss rate were still

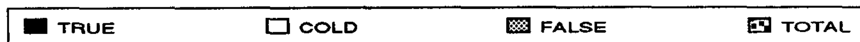
Figure 6: Effect of invalidation scheduling on the miss rate (%).



(a) Block size=64 bytes



(b) Block size=1,024 bytes



possible beyond the reduction provided by WBWI.

The simulations for B=1,024 show that further improvement is possible because the best protocol (SRD) does not always reach the essential miss rate of the trace, especially in the cases of LU and MP3D. Because of the discrepancy between the miss rates of WBWI and MIN (but not between RD and WBWI), it appears that any improvement will have to deal with the problem of block ownership. This line of thought leads to systems with multiple block sizes [7], or even systems in which coherence is maintained on individual words.

The MAX scheduling of invalidations always yields more misses than any other schedule, including OTF. For smaller block sizes the worst-case schedule gave a miss rate almost equal to OTF: because of the small block size, there are few opportunities to create ping-ponging among multiple processors accessing the same block at the same time. However, for larger block sizes, the worst-case miss rate may be significantly higher than OTF. Delaying invalidations at the receiving end should also help avoid worst-case situations such as the one in MAX.

We were able to run some simulations for the larger data set sizes namely for LU200, MP3D10000 and WATER288. In these traces, the effect of false sharing moves to larger block sizes. We do not show the curves for lack of space. The reader can find more details in [10]. The effects of invalidation schedules are similar to the case of small data sets, but these effects are much reduced for B=64 since the difference between the on-the-fly miss rate and the essential miss rate is always less than 20%. For B=1,024, the false sharing components are very large and the protocols are still quite far from the essential miss rate, as in the case of the smaller data set sizes. We have also observed a very large miss rate for MAX in the case of LU.

8.0 Conclusion

In this paper, we have introduced a classification of multiprocessor cache misses based on the fundamental concept of interprocessor communication. We have defined essential misses as the minimum number of misses for a given trace and a given block size so that the trace receives all the data inputs that it needs to execute correctly. Essential misses include cold misses, which communicate initial values to the processor, and true sharing misses, which communicate updates from other processors. The rest of the misses are useless misses in the sense that the trace would execute correctly even if they (or alternatively the invalidations leading to them) were not executed in the cache.

We have shown that previous classifications tend to overestimate the amount of false sharing. For the small data set sizes (fine granularity of parallelism) and the four benchmarks the false sharing component is significant even for 16-byte blocks. For larger data sets, false sharing effects are moved to larger blocks.

We have also simulated several approaches to effectively detect and eliminate useless misses by dynamically delaying and combining invalidations. For B=64 (caches), all these techniques were very effective, except for the pure send delayed protocol; there is little room for improvement. Considering hardware complexity, protocols with partial block invalidations such as WBWI may be preferable for small block sizes whereas a pure receive delayed protocol is probably preferable for large block sizes. For B=1,024, the techniques addressed in this paper are particularly needed. In this case, delaying and combining invalidations at both the sending and receiving end is useful but not sufficient to remove all useless misses.The main reason is the need to maintain ownership.

We have not displayed the amount of memory traffic generated by these protocols. The protocols with reduced miss rates also have reduced miss traffic. However, the traffic is very high for large block sizes. At this level of traffic, delayed write-broadcast or delayed protocols with competitive updates, which can reduce the number of essential misses, may become attractive.

Finally, the current classification is applicable to infinite caches only. However, it can easily be extended to finite caches by introducing replacement misses. A replacement miss is an essential miss since the value is needed to execute the program. Coherence misses can then be classified into PFS and PTS misses according to the algorithm in this paper. We expect that the fraction of essential misses will increase in systems with finite caches. This effect will depend on the cache size, cache organization and replacement policy.

9.0 References

[1] Bennett, J.K., Carter, J.B., and Zwaenepoel, W., "Adaptive Software Cache Management for Distributed Shared Memory Architectures," *Proc. of the 17th Ann. Int. Symp. on Comp. Arch.*, pp. 125-134, Jun. 1990.

[2] Borrmann, L., and Herdieckerhoff, M., "A Coherency Model for Virtual Shared Memory," *Proc. of Int. Conf. on Parallel Proc.*, Vol. 2, pp.252-257, Jun. 1990.

[3] Boyle, J., et al., "Portable Programs for Parallel Processors". Holt, Rinehart, and Winston Inc.,1987.

[4] Brorsson, M., Dahlgren, F., Nilsson, H., and Stenström, P.,"The CacheMire Test Bench — A Flexible and Effective Approach for Simulation of Multiprocessors," *Proc. of the 26th Annual Simulation Symposium*, March 1993.

[5] Censier, L.M., and Feautrier, P., "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. on Comp.*, Vol. C-27, No. 12, pp. 1112-1118, Dec. 1978.

[6] Chen, Y-S, and Dubois, M., "Cache Protocols with Partial Block Invalidations," *Int. Symp. on Parallel Proc.*, Apr. 1993.

[7] Dubnicki, C., and LeBlanc, T.J.,"Adjustable Block Size Coherent Caches," *Proc. of the 19th Ann. Int. Symp. on Comp. Arch.*, pp. 170-180, May 1991.

[8] Dubois, M., Barroso, L., Wang, J.C., and Chen, Y.S., "Delayed Consistency and its Effects on the Miss Rate of Parallel Programs," *Supercomputing'91*, pp. 197-206, Nov. 1991.

[9] Dubois, M., and Scheurich, C., "Memory Access Dependencies in Shared Memory Multiprocessors," *IEEE Trans. on Soft. Eng.*, 16(6), pp. 660-674, Jun. 1990.

[10] Dubois, M., Skeppstedt, J., Ricciulli, L., Ramamurthy, K., and Stenström, P., "The Detection and Elimination of Useless Misses in Multiprocessors," USC Tech. Rep. No. CENG-93-2, Jan.1993.

[11] Eggers, S. J., and Jeremiassen, T. E., "Eliminating False Sharing," *Proc. of the 1991 Int. Conf. on Par. Proc.*, pp. I-377-I-381, Aug. 1991. Also published as TR 90-12-01, Univ. of Washington, Dept. of Comp. Sc. and Eng, Seattle, Washington.

[12] Ekstrand, M., "Parallel Applications for Architectural Evaluations of Shared-Memory Multiprocessors." Master's thesis, Dept. of Comp. Eng., Lund Univ., Sweden, Feb. 1993.

[13] Lenoski, D., Laudon, J.P., Gharachorloo, K., Gupta, A., and Hennessy, J.L.,"The Directory-based Cache Coherence Protocol for the DASH Multiprocessor," *Proc. of the 17th Ann. Int. Symp. on Comp. Arch.*, pp. 148-159, Jun. 1990.

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

[15] Stenström, P., "A Survey of Cache Coherence Schemes for Multiprocessors," *IEEE Computer*, Vol. 23, No. 6, pp. 12-24, Jun. 1990.

[16] Torrellas, J., Lam, M.S., and Hennessy, J.L., "Shared Data Placement Optimizations to Reduce Multiprocessor Cache Misses," *Proc. of the 1990 Int. Conf. on Parallel Proc.*, pp. 266-270, Aug 1990. Also published as "Measurement, Analysis, and Improvement of the Cache Behavior of Shared Data in Cache Coherent Multiprocessors" Tech. Rep. CSL-TR-90-412, Stanford University, Stanford, CA, Feb. 1990.

Appendix A — Classification Algorithm

A miss is classified at the end of the lifetime of the block (either at the time it is invalidated or at the end of the simulation). The classification algorithm uses the following flags associated with each processor: one Presence flag (P), one Essential Miss flag (EM), and one First Reference flag (FR) per block plus one communication flag (C) per word. We specify the actions that are needed for each load (`read_action`) and store (`write_action`) in the trace and actions taken at the end of the simulation (`end_of_simulation`).

Below we show Pascal-like code for the algorithm. N denotes the number of processors, T and F denote logical True and False, and `block_ad` and `word_ad` denote the block and word address, respectively:

```
read_action(proc_id,block_ad,word_ad):
 if not P[block_ad,proc_id]then
  begin
   EM[block_ad,proc_id]:=F;
   P[block_ad,proc_id]:=T;
  end;
 if C[word_ad, proc_id]then
  begin
   EM[block_ad, proc_id]:=T;
   for i:=0 until block_len-1 do
     C[block_ad*block_len+i,proc_id]:=F;
  end;

write_action(proc_id, block_ad,word_ad):
 read_action(proc_id,block_ad,word_ad);
 classify(proc_id,block_id,F);
 for i:=0 until N-1 do
   if i<>proc_id then
     begin
      C[word_ad,i]:=T;
      P[block_ad,i]:=F;
     end;

classify(proc_id,block_ad,my_block):
 for i:=0 to N-1 do
  if P[block_ad,i]and
      (my_block or (i<>proc_id)) then
   begin
    if not FR[block_ad,i] then CM:=CM+1;
     else if EM[block_ad,i] then PTS:=PTS+1;
      else PFS:=PFS+1;
    FR[block_ad,i]:=T;
   end;

end_of_simulation:
  for "each processor i and block j" do
    classify(i,j,T);
```