THE DETECTION AND ELIMINATION OF USELESS MISSES IN MULTIPROCESSORS

Michel Dubois, Jonas Skeppstedt*, Livio Ricciulli,

Krishnan Ramamurthy, and Per Stenström*

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA90089-2562, U.S.A.
(213)740-4475
dubois@paris.usc.edu

*Department of Computer Engineering
Lund University
P.O. Box 118, S-221 00 LUND, Sweden
46-46-107-523
per@dit.lth.se

Abstract

In this paper we introduce a classification of misses in shared-memory multiprocessors based on inter processor 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 program execution.

Based on the new classification we evaluate miss reduction techniques in hardware, based on *delaying* and *combining* invalidations. We compare the effectiveness of five different protocols for combining invalidations leading to useless misses for cache-based multiprocessors and for multiprocessors with virtual shared memory. In cache based systems these techniques are very effective and lead to miss rates which are close to the minimum. In virtual shared memory systems, the techniques are also effective but leave room for additional improvements.

P.O.C: Michel Dubois

Poster Session: We do not want the paper to be considered for a poster session.

Keywords: Shared memory multiprocessor, distributed shared memory, cache coherence, consistency models, performance evaluation

THE DETECTION AND ELIMINATION OF USELESS MISSES IN MULTIPROCESSORS

Abstract

Future shared memory multiprocessors will include features to hide the increasing performance gap between memories and processors. Usually, stores can be overlapped with the local execution of the processor. This overlap may increase the miss rate, as we will show in this paper, but it also offers new opportunities to eliminate useless misses and therefore to reduce the miss penalty as well. Attempts have been made in the past to detect useless misses and to eliminate them by hardware or compiler approaches.

In this paper we introduce a classification of misses in shared-memory multiprocessors based on inter processor 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 or false sharing misses, and can be ignored without affecting program execution. Previous classifications did not detect useless misses correctly and tended to overestimate their number.

Based on the new classification we evaluate miss reduction techniques in hardware, based on *delaying* and *combining* invalidations. We compare the effectiveness of five different protocols for combining invalidations leading to useless misses for cache-based multiprocessors and for multiprocessors with virtual shared memory. In cache based systems these techniques are very effective and lead to miss rates which are close to the minimum. In virtual shared memory systems, the techniques are also effective but still leave room for improvements.

1. 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: systems with private caches [7,21] and systems with virtual shared memory [1,3,17], in which multiple copies of the same data may exist and coherence must be maintained among them. There are some implementation differences between the two types of systems. The transfer of data between caches and memories is done in blocks under hardware control and the block size is typically less than 256 bytes; in a system with virtual shared memory data transfers are managed in software and involve pages of size typically larger than 512 bytes. In this paper we consider block sizes between 4 and 4096 bytes and therefore we address both kinds of systems. Block sizes between 4 bytes and 256 bytes are referred to as the *caching range*, while block sizes between 512 bytes and 4096 bytes are referred to as the *paging range*.

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. Therefore, there are two types of memory requests in a system with a write-invalidate protocol: invalidations of remote copies and misses. 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. Penalties reduce the processors' efficiency and this effect becomes worse as more powerful processors are designed and as more processors are supported. If we ever want to build truly scalable shared memory multiprocessors, we need to reduce memory penalties to a minimum. Current research on memory consistency models is key to hiding the large access latencies of shared-memory.

Whereas invalidation penalties can be easily eliminated through more aggressive consistency models [11,18], load miss latencies are much harder to hide. In general, 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). Very large (i.e., virtually infinite) caches can eliminate replacement misses. Cold misses can be dealt with by prefetching and preloading and by using larger block sizes. Effective prefetching requires significant compiler support, and its feasibility has only been demonstrated so far for scientific programs. Coherence misses are difficult to hide by prefetching. Bigger block sizes may increase the number of coherence misses, because of *false sharing* [12,22]. Loosely speaking false sharing is the sharing of block without actual sharing of data. False sharing misses are the consequence of having different processors accessing different words in the same block, as opposed to true sharing misses, which are caused by the sharing of data items. False sharing misses are useless misses in the sense that they do not bring new values in cache.

Current classifications of multiprocessor misses fail to identify the set of useless misses. Our first contribution is to revise the current classifications of misses in order to obtain more accurate numbers for the different components of the miss rate. This new classification, introduced in Section 2, shows that true sharing miss rates are underestimated and that false sharing miss rates are overestimated with current counting techniques [12,22]. In the process we will define the set of essential misses which is the set of misses required for a correct execution. This new classification is important in general because we cannot evaluate and compare competitive approaches to reducing miss rates unless we have good measures of miss rate components. For example, in compiler-based approaches to miss reduction it is important to understand how much improvement in the miss rate was due to the reduction of false sharing and how much was due to better locality. Moreover, by identifying the minimum possible miss rate for a given execution, we can understand how close we are to the minimum miss rate and whether further improvements are possible.

Aggressive techniques to tolerate memory latency tend to change the timings of invalidations. For example, in the DASH machine [18], 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. This paper focuses on the effects of the timing of invalidations on the miss rate of a program's execution. We only consider pure write invalidate protocols with a single block size. We compare the effectiveness of techniques that have been proposed to reduce the miss rate on shared data

accesses in both the caching and paging ranges. The general approach is to delay invalidations and to eliminate invalidations leading to useless misses. We show an optimum invalidation schedule, which yields the minimum possible miss rate (i.e., the *essential* miss rate). This minimum is reached by sending all invalidations to all processors with a copy immediately, and then delaying and combining all invalidations received by a processor until a stale value is accessed. 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 and which, in some cases, causes a large number of misses.

Section 2 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 present and analyze our simulation results and we then conclude.

2. CLASSIFICATION OF MISSES IN MULTIPROCESSORS

Mark Hill proposed a classification of misses in uniprocessor caches [16]. Misses are categorized into compulsory, capacity and conflict misses. Such a classification is useful because it helps explain the effects of a design decision on the miss rate. The same type of classification is sorely needed for multiprocessor caches. In a multiprocessor, misses can be classified as cold, replacement and coherence misses. It is useful to refine this classification to include false and true sharing misses. Unfortunately, there is no agreement in the current literature about what a false sharing miss should be. No fundamental definition of a false sharing miss has been given; current proposals to detect and count cold and coherence misses, by Eggers and Jeremiassen [12] and by Torrellas, Lam and Hennessy [22][1] yield different outcomes.

Another drawback of current proposals is that true sharing misses are not directly related to the notion of data communication among processes in a parallel computation. Intuitively, true sharing misses should be the minimum set of misses communicating new values from other processors; they are essential in the sense that the processor would read a stale value and the execution would not be correct if they were not executed. Similarly, the set of false sharing misses should be the maximum set of misses such that the program would still execute correctly if they were ignored (i.e. if the invalidation leading to them had not been executed in the cache) In the following, we show that it is possible to detect these true and false sharing misses and then we compare the miss counts obtained with the new detection technique with the miss counts obtained with current techniques. We limit the discussion to infinite cache sizes.

2.1 Torrellas's Scheme

A cold sharing miss in a trace for a block size of B words is detected on a miss if the accessed *word* is referenced for the first time by a given processor. Therefore the same block can have up to B cold misses in each processor during the simulation. A true sharing miss is detected on a miss such that the referenced word is not accessed for the first time and such that it is also a miss in a system with a block size of one. In practice two simulations are run: the first one is a simulation of the target system, which can be a trace-driven or execution driven simulation, with a block size of B words; the second one is driven by the trace of references derived from the first

1. In the following we will refer to these two proposed classifications as Eggers's and Torrellas's.

simulation to detect misses in a system with a block size of one word. Any miss in the first simulation is classified as a cold miss, a true sharing miss or a false sharing miss if it is a cold miss, a warm miss or a hit in the second simulation. The intuition behind this classification is that a miss in a system with a block size of one word is always a cold miss or a true sharing miss and is essential because without the value, the processor cannot finish its task correctly.

There are several drawbacks to this approach to classifying misses. First of all, the way cold misses are detected is not conventional nor consistent with approaches adopted in other papers. Usually a cold miss is counted on the first access to a given block by a given processor. As a result, the classification is not applicable to algorithms in which there are large numbers of cold misses. It is only targeted to iterative algorithms in which each word is accessed more than once. Unfortunately, this removes some useful parallel algorithms from consideration. One example is the non-shuffling FFT (see Section 4) which has only cold misses for a block size of one word. Another one is matrix multiply. Consider the multiplication of two *NxN* matrices *A* and *B* storing the result into a matrix *C*, shown in Fig. 1.

Fig. 1 Program for Matrix Multiplication

```
For i=1,N
   For j=1,N
      {temp=0;
       For k=1,N
          temp=temp+A[i,k]*B[k,j];
       C[i,j]=temp;}
```

The only writable data in this program are the elements of the resultant matrix *C*. Assume for the sake of argument that the matrices are stored row-wise in the shared-memory, that two processors compute each half the columns of *C*, i.e. a submatrix *NxN/2* of *C*, and that the block size is equal to *N* data elements. False sharing misses are likely to result as each processor modifies *C* in turns. However, since all misses on *C* are cold misses in the system with a block size of one word, the misses are all classified as cold misses.

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 following sequence in Fig. 2, where `Load i` and `Store i` correspond to a load and store in word i of the same block and different lines correspond to successive references in the trace (we will comment on columns *Eggers* and *Correct* later.)

Fig.2. Sequence showing the fundamental shortcoming of current classifications
CM: Cold Miss; FSM: False Sharing Miss; TSM: True Sharing Miss
Words 1 and 2 are in the same block.

Ref.	P1	P2	Torrellas	Eggers	Correct
T0:		Load 1	CM	CM	CM
T1:	Load 2		CM	CM	CM
T2:	Load 1		-	-	-
T3:		Store 1	-	-	-
T4:	Load 2		FSM	FSM	TSM
T5:	Load 1		-	-	-

The miss at reference T4 brings a new value defined at T3 and read at T5 in the cache of

processor P1, and yet it is classified as a false sharing miss. If we did not execute the miss at T4 (or equivalently ignored the invalidation at T3) and kept the old block in the cache instead P1 would read a stale value at T5. Initial word values and values communicated between two processors by a sequence of invalidation/read miss in the system with a block size of one are inputs to the computation of a processor. Torrellas classifies a reference as a false sharing miss if the *reference causing the miss* is not an input to the computation. In their paper Torrellas et al. introduce the notion of *prefetching effects*; however they do not attempt to quantify these effects.

2.2 Eggers' Scheme

Cold misses happen 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 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.

If we ignore the discrepancy between the definitions of cold misses, Eggers's scheme counts more false sharing misses and less true sharing misses than Torrellas's because any true sharing miss in Eggers's classification must also be a true sharing miss in Torrellas's. It is not difficult to find sequences such that more true sharing misses are counted in Torrellas's method. The sequence in Fig. 3 shows such an occurrence as well as the discrepancy in the classification of cold misses. The correct classification shows the shortcomings of both classification schemes.

Fig. 3. Sequence showing the differences between Eggers's and Torrellas's classifications
CM: Cold Miss; FSM: False Sharing Miss; TSM: True Sharing Misses
Words 1 and 2 are in the same block.

Ref.	P1	P2	Torrellas	Eggers	Correct
T0:	Load 2		CM	CM	CM
T1:		Load 1	CM	CM	CM
T2:		Store 2	-	-	-
T3:	Load 1		CM	FSM	FSM
T4:		Store 1	-	-	-
T5:	Load 2		TSM	FSM	TSM
T6:	Load 1		-	-	-

2.3 Correct Classification

The two existing schemes for classifying invalidation misses count a true sharing miss only when the reference causing the miss accesses a new value communicated to the process. These values can be defined by a store from another processor at any time in the past for Torrellas's scheme while Eggers scheme considers only values defined since the invalidation causing the miss.

However, values can also be communicated after the occurrence of the miss. In Fig. 2, the value modified by P2 at T3 is read by P1 at T5. Both current classification schemes incorrectly classify the miss in P1 at T4 as a false sharing miss. Therefore, both schemes tend to overestimate false sharing and underestimate true sharing. It is important to classify misses correctly so that the

proper remedy can be applied to remove them. Clearly, it is impossible to eliminate misses that communicate inputs to each individual process.

To explain the miss classification algorithm we first need some definitions. Consider a given block. The *lifetime* of a block in the cache following a miss is the time between the miss bringing the block into the cache and the invalidation removing it from the cache. When a processor modifies a word in the block it *sends* a new value to all other processors. These processors may *receive* the value when they access it for the first time. Besides values received from other processors the processor also needs to receive the initial value of the data in the block. Cold misses and true sharing misses form the set of *essential* misses. Values are *communicated* to a processor when they are loaded into the cache on an essential miss. These definitions lead to the following classification of misses in a system with infinite caches.

Essential miss: A miss is an essential miss if, during the lifetime of the block in the cache, a processor reads for the first time a value defined since the previous essential miss. The first miss to the block by one processor is also an essential miss.

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

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

Pure False Sharing miss (PFS): A non-essential 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. Any store by other processors into the block following this cold miss sends new values. The processor may never receive any of these values. The first true sharing miss occurs when the processor receives one of these values during the lifetime of the block in the cache; at this time all values defined since the initial cold miss are also communicated to the processor. Note that between the initial cold miss and the first true sharing miss there may have been several false sharing misses; during the lifetime of the block following these false sharing misses no value was received which was sent after the initial cold miss. Therefore 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 value loaded on the cold miss instead. True sharing misses can be detected one after the other by detecting first accesses to values that were modified since the previous essential miss.

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

Cold and True Sharing miss (CTS): Cold miss which communicates a value defined since the start of the simulation and received during the lifetime of the block in the cache.

Cold and False Sharing miss (CFS): Cold miss which communicates a value defined since the start of the simulation but not received during the lifetime of the block in the cache.

Pure Cold miss (PC): Cold miss which only communicates initial values (i.e., it is not preceded in the trace by any modification by other processors.)

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.

In a system with finite caches PTS and PFS misses can become replacement misses. Also new misses are introduced because of replacements. Since we do not use finite caches in this study, we will not pursue this classification further.

Fig.4 High-level specification of the classification algorithm into PTS, CTS, PC, CFS, and PFS misses P is the number of processors

```
For each word: C-flag vector: P binary flags, initially reset
For each block:EM-flag vector: P binary flags
               FR-flag vector: P binary flags, initially reset

Actions taken on each read:
  Upon a read miss:
    The EM-flag is reset
  Always:
    If the C-flag of the accessed word is set,
      the EM-flag for the block is set, and
      all C-flags for the words in the block and processor are reset

Actions taken on each write:
  All actions performed on a read, followed by:
  If the copy is shared do:
    For each valid block copy (in cache i):        **classification**
      If all C-flags of the block for processor i are reset and
        if the FR-flag and the EM-flag are reset then PC++
      If any C-flag for processor i is set and
        if the EM-flag is reset and the FR-flag is reset then CFS++
      If the EM-flag is set and the FR-flag is reset then CTS++
      If the FR-flag and the EM-flag are set then PTS++
      If the FR-flag is set and the EM-flag is reset then PFS++
      Set the FR-flag of processor i
  Always:
    The C-flags for the word and for all processors are set
    The C-flag for the writing processor is reset

Actions taken at the end of the simulation:
  Scan all block frames and classify the misses of all valid
    blocks according to the **classification**
```

2.4 Classification Algorithm

One communication flag (C-flag) is associated with each word in memory and each processor. When a C-flag is set, the latest value of the word has not been communicated to the processor. Whenever a processor modifies a word it sets all other processors' C-flags for the word. An essential miss flag (EM-flag) per cache blockframe detects that a word with the C-flag set has been read during the lifetime of the block in the cache. The EM-flag is set and the C-flags of a block are reset in a processor at the time the new value is received by the processor. The classification is done based on the value of the EM-flag at the time when a block is invalidated; moreover, at the end of the simulation, the caches are scanned and the classification is done on all the remaining valid block.

The algorithm also uses a first reference flag (FR-flag) per block and per processor to detect and classify cold misses. A high level description of the classification algorithm is shown in Fig. 4. PTS, CTS, PC, CFS and PFS are counters counting misses in the different miss classes.

2.4 Effect of Block Size on Essential Miss Counts

Essential misses form the set of necessary and sufficient misses for a correct execution. One interesting observation is that the number of essential misses observed in a trace cannot increase when the block size increases. 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 ith essential miss for a system with block size B_1 must happen before the ith essential miss for a system with block size B_2, because each miss with a block size B_2 brings more values into the cache and the number of communicated values after any number of references cannot be less in the system with block size B_2.

Similarly the number of cold misses cannot increase with the block size because more values are brought in on every miss in a system with a larger block size.

Fig. 5. Sequence showing that the number of PTS misses can increase with the block size
Words 0 and 1 are in the same block of size 2.

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

The number of pure true sharing misses decreases also with the block size in general, but this is by no means certain. In Fig. 5, 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 pure true sharing misses increases. Some CTS misses may become PTS misses when the block size increases. However, the total number of CTS and PTS misses cannot grow when the block size increases, for the same reason as for the essential misses, i.e. more values are communicated at each miss when the block size increases.

2.5 Detection and Elimination of Useless Misses through Hardware

It should be clear that it is impossible to eliminate essential misses in a pure write invalidate protocol. On the other hand, designing a hardware protocol which detects and eliminates useless misses as defined in this paper is not difficult. In this Section we wish to introduce a simple protocol which totally eliminates useless misses. We do not claim that this protocol is efficient or should be implemented. We simply want to show that the new classification of misses leads to clearer understanding of how to eliminate them. We also want to show a general approach for designs tolerating some false sharing in order to reduce the overhead of detection.

We need to avoid executing in the cache any invalidation leading to a false sharing miss. This means that we should not invalidate the block until the processor attempts to read a stale value. This can be done if the protocol is write though. Instead of invalidating the block every

time an external write hits in the cache, we buffer the address of the modified word in an invalidation buffer and invalidate the block when a local access is made to a word whose address is present in the buffer. The miss immediately follows the invalidation with an access to the stale word and therefore is a PTS miss (actually it is a true sharing miss in any of the three classifications.) The invalidation buffer could also be implemented with a dirty bit associated with each word in each block of the cache. Note that this implementation "mimics" the essential miss detection algorithm that we have proposed in Section 2.3 and its miss rate is the essential miss rate of the trace.

Fig. 6. Sequences showing that the number of essential misses depends on the interleaving of the trace
Words 0 and 1 are in the same block.

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

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

2.6 Invalidation Delaying and Combining

There are many ways to improve the efficiency of the write-through protocol of the previous section but we are only interested in the idea behind the approach: 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. Consecutive stores to the same block can be delayed and combined in the sending processor so that a single set of invalidations is sent for all the combined stores. Delaying the sending of a store per se does not help. It can increase the false sharing miss rate when the store is delayed across an essential miss in the receiving processor (without the delay, it would have ben 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 we show in Fig. 6. Delaying stores at the sending end can only help if the delays lead to combining of invalidations. Delaying at the receiving end is never harmful to the miss rate, because optimum combining of invalidations from any processor can take place, by simply ignoring the invalidations until one of them causes an essential miss.

The essential miss rate--even in its more fundamental form introduced in this paper-- is not an intrinsic property of an application, as previously believed [22]: it is only a property of a particular execution (or a particular interleaved trace) and is timing dependent, as is shown by the sequences in Fig. 6. The two sequences are equivalent executions but the second one yields less essential misses, in any existing classification. That sequence also shows that delaying stores may in some cases increase the essential miss rate.

2.7 Comparison Between the Classification Schemes

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 of the benchmarks runs with the larger data set sizes (see Section 4), namely JACOBI64, WATER288, LU200 and MP3D10000 both in the caching and in the paging ranges. As can be seen from Table 1, current measures of false and true sharing are totally unreliable. Eggers's scheme tends to exaggerate the amount of false sharing and to underestimate true sharing, because the classification of a miss is done at the reference that misses and ignores the possibility of communicating new values in subsequent references. In some cases the measures are off by one order of magnitude.

Table 1: Comparison between the three classification schemes

BENCHMARKS	JACOBI	JACOBI	WATER	WATER	LU	LU	MP3D	MP3D
BLOCK SIZES	32	1024	32	1024	32	1024	32	1024
PTS-CORRECT	21,352	7,048	394,256	93,384	5,769	7,941	188,120	82,125
PTS-EGGERS	19,709	980	393,810	68,145	2,845	2,558	178,206	67,447
(error in %)	-7.7	-86.1	-0.1	-27.0	-50.7	-67.8	-5.3	-17.9
PTS-TORRELLAS	19,743	1,044	393,788	73,085	597	183	177,272	112,562
(error in %)	-7.5	-85.2	-0.1	-27.1	-89.6	-97.7	-5.8	+37.1
COLD-CORRECT	5,216	396	24,034	1,837	110,955	5,545	46,242	4,058
COLD-TORRELLAS	5,257	4,206	24,122	2,756	113,812	9,827	52,264	26,011
PFS-CORRECT	22,987	154,991	5,712	91,297	11,839	79,882	31,206	266,245
PFS-EGGERS	24,630	161,059	6,158	116,536	14,763	85,265	41,120	280,923
(error in %)	+7.1	+3.9	+7.8	+27.6	+24.7	+6.7	+31.8	+5.5
PFS-TORRELLAS	24,555	157,185	6,092	110,677	14,154	83,358	36,032	213,855
(error in %)	+6.8	+1.4	+6.6	+21.2	+19.5	+4.3	+15.5	-19.7

Errors in the detection technique used by Torrellas are less consistent. Torrellas's scheme tends to underestimate true sharing for the same reason as Eggers's does. However, this effect can be compensated by the fact that a new value can be loaded in cache on several true sharing misses and still cause a true sharing miss itself. The sequence in Fig.7 illustrates this point. Since the true sharing misses are the minimum number of misses needed to communicate new values to each process, the correct classification considers that the new values of words 1 and 2 defined at times T2 and T3 by P2 are communicated to P1 at T4. In Torrellas's scheme however, the value of word 1 is still considered communicated to P1 at time T6, causing an additional PTS miss. Another problem with Torrellas's classification is the effect of classifying a large number of PTS and PFS misses as PC misses. Overall the cumulative effects of these three discrepancies are impossible to predict. The numbers in Table 1 indicate that in general the net effect tends to be an overestimation of the number of false sharing misses.

Fig. 7. Sequence showing overestimation of PTS misses in Torrellas's scheme
Words 1, 2, and 3 are in the same block.

Ref.	P1	P2	Torrellas	Eggers	Correct
T0:	Load 1		PC	PC	PC
T1:	Load 2		-	-	-
T2:		Store 1	PC	PC	PC
T3:		Store 2	-	-	-
T4:	Load 2		PTS	PTS	PTS
T5:		Store 3	-	-	-
T6:	Load 1		PTS	PFS	PFS

3. SCHEDULING OF INVALIDATIONS

Previous studies have shown that delaying invalidations reduces the miss rate and the traffic for shared blocks [10,17]. The protocol of Section 2.5 demonstrates that the essential miss rate is reachable if the protocol uses write-through caches and invalidations are combined in an invalidation buffer until the processor accesses a word of the block with an invalidation pending in the buffer. Unfortunately, write-through caches generate an unacceptable amount of write traffic. To improve the protocol we need to add ownership and make it write back. Ownership has its costs in terms of miss rate. At any one time there can be only one writer for the block and the writer must have the latest copy of all the words of the block so that it can provide the valid block copy when a miss occurs in a different processor. Loads are not affected, but a store miss must be executed every time a store accesses a non-owned block with a pending invalidation for ANY one of its words. These additional misses are the performance cost of enforcing ownership.

Another performance cost is the detection of stale words. The protocol in Section 2.5 required one dirty bit per word in every blockframe of each cache. This is acceptable for cache-based systems with small block sizes but it may not be acceptable for page-sized blocks in virtual shared memory systems. Another approach to detecting potential stale words relies on synchronizations. The sending of invalidations can be delayed and combined until the next *release* and received invalidations can be delayed and combined until the next successful *acquire*. In this case, there may be additional performance costs. First of all, the effect of delaying the sending of stores is mostly unpredictable unless it leads to the combining of several invalidations. In this case, the delaying and combining of stores lessen the performance cost of enforcing ownership. Another problem is that the acquire causing the invalidation may not be close enough to the reference triggering the essential miss in the write-through protocol; in fact, the acquire could be totally unrelated to that word. The resulting additional false sharing misses are the performance cost of relying on synchronization to prevent the reading of stale words.

We have simulated various schedules of invalidations, in order to understand their effects on the miss rate. We also adopt the terminology introduced in [10]. Similar protocols have been published under different names in [1,3,17]. They are: MIN, OTF, RD, SD, SRD, WBWI, and MAX.

MIN: Write-through with Word Invalidation

This is the ideal write-through protocol of Section 2.5. 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

Processors execute their store without delay. Whenever a processor executes a store to a word of the block, it must be an owner and must have the latest copy of all words of the block. Invalidations are propagated 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. There are two reasons why the hit rate of a received delayed protocol is not as high as the hit rate of MIN: invalidation time is based on the time at which an acquire is executed and ownership must be enforced. To avoid these two problems we can try to delay the sending of the invalidation.

SD: Send Delayed Protocol

Each processor has a buffer for sending invalidations. This buffer can be a write cache similar to the one described in [6]. A write cache is similar to a write buffer but contains entire blocks with 1 dirty bit per word to signal modified words. If the processor is the owner at the time of the store, the store is completed without delay. Otherwise, the store is done in the write cache. Stores for the same block combine in the write cache, possibly reducing the number of invalidations sent. On a replacement in the write cache, ownership is acquired for the replaced block. The latest time to remove a block from the write cache is at the execution of a *release*. When an invalidation is received, it is executed immediately in the cache. The effect of delaying stores on the miss rate is unpredictable, but it usually helps because of store combining. When we reach a store in the simulation of the SD protocol, we first check for ownership. If the cache owns the block the store is executed immediately; otherwise it is inserted in a store cache of infinite size. In essence, we apply the OTF protocol on a modified trace: the delayed stores are moved in FIFO order to a location in the trace directly preceding the next release by the processor.

SRD: Send and Receive Delayed Protocol.

Each processor has a buffer for received invalidations and one for sending invalidations. The buffer for sending invalidations can be a write cache. A store is buffered in the write cache if the processor is not an owner. At each *release*, the write cache must be flushed. Ownership is acquired for each block replaced in the write cache. The invalidations are inserted in a buffer for receiving invalidations associated with each processor. This buffer is flushed whenever an *acquire* is executed. There is combining of invalidations in both buffers. In essence, in the simulation, we apply an RD protocol on a modified trace in which stores on clean and stores misses are moved down in the trace to the next release by the processor.

WBWI: Write-back with Word Invalidate Protocol.

This is basically the MIN algorithm, but with ownership to reduce the write traffic. On a

miss, entire blocks are loaded in cache, but invalidations are done on word, with the help of an additional dirty bit per word. Stores are executed and their invalidations are propagated on-the-fly, in the order of the trace the first time a word is modified in a cached block. A load miss is triggered when it accesses a dirty word. A store miss is triggered if the processor does not have ownership and if ANY word in the block is dirty (this is the cost of maintaining ownership.) WBWI is also similar to RD except that it relies on the dirty bit 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. The definition of release consistency that we adopt is the strict definition of the DASH: Stores of 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. Each processor has a store buffer for pending stores. A store is pending if its invalidations have not been scheduled. If a store misses in a cache, then it is executed immediately in the cache and its invalidations are scheduled. Otherwise, it is buffered in a store buffer. On a read hit we first check to see if there is a pending store in a different processor with the same address. In case there are several stores pending in different processors we pick a store that causes a miss in its processor cache, if there is one. At a release in a processor we propagate all the invalidations for all the pending stores in the processor.

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

4. 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. The effects due to altered executions are related to the particular timings of a simulated machine and are impossible to separate from the effects of invalidation scheduling. An interleaved memory access trace, on the other hand, gives us a fixed source of references in a fixed order to compare invalidation schedules independently of any other influence. Note that we never violate the dependencies in the trace when we apply different schedules of invalidations in our simulations. Therefore the trace never becomes absurd, as may happen in other studies [2]. We have collected traces from six benchmark programs and two different data set sizes. All benchmarks were run for 16 processors and infinite cache sizes.

The first three benchmarks are parallel applications developed at Stanford University (MP3D, Water, and LU) of which the first two are also contained in the SPLASH suite [20]. These applications are written in C using the Argonne National Laboratory macro package [4]. and compiled with the `gcc` compiler (version 2.0) using optimization level -O2. Traces from these benchmarks have been captured by the CacheMire Test Bench, a tracing and simulation tool for shared-memory multiprocessors [5].

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. The particles are represented by an array of particle objects, each of which is 36 bytes.They are statically allocated to processors in an interleaved fashion. 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 other particle. We have run MP3D with 1,000 particles for 20 time step (referred to as MP3D1000) and with 10,000 for 10 time steps (referred to as MP3D10000). In both cases, we run MP3D with the locking option switched on.

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. The molecules are represented by objects, each of which is 680 bytes, and are statically allocated to processors in a coarse-grained fashion. 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 four 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 the left after these columns have been modified themselves. Columns 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).

The next two traces were produced by the Prism Simulator [13], an execution-driven simulator. ModulaP [14] source code is compiled to produce intermediate C code which consists of a number of slices (or procedures) which represent the atomic actions performed in turn by the different threads. The intermediate C code produced by the ModulaP compiler is lexically processed to insert statements which output memory reference activity. This modified C code is linked to the Prism simulator kernel. We allow the simulator to run with as many processors as threads to avoid thread migration and simplify the analysis. The two benchmarks are NSFFT and QSORT.

NSFFT is a parallel implementation of the Discrete Fourier Transform on an array of 16-byte complex floating-point numbers [8]. The workload is partitioned by assigning an equal part of the array to each processor. Each processor reads values from other processors' partitions and then updates its partition with these new values. We have run NSFFT with 512 complex numbers.

QSORT is a parallel divide and conquer sorting algorithm [19]. It is synchronized through a dynamic job queue. Each process obtains exclusive access to a share of the array to sort, splits it into two sub array and dynamically posts unsorted portions to the job queue. The algorithm terminates when all processors have terminated their job and the job queue is empty. There are two serialization points in the algorithm where the job queue is updated. We have run QSORT on arrays of 5,000 randomly selected 32-bit integers.

Finally, **JACOBI** was written by us using the ANL macros [4] provided with the SPLASH benchmark suite. JACOBI is an iterative algorithm for solving partial differential equations [24]. Two 64x64 grid arrays of 8 bytes double precision floating point numbers 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. Each of the 16 processors is assigned to the update of a square 16x16 subgrid. After each iteration, the processors synchronize through a barrier synchronization, a test for convergence is done and the two arrays are switched. Therefore, 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.

Since trace-driven simulations are time consuming, we cannot afford to study traces from executions on real machines with a large number of processors with appropriately large data set. We see no simple answer to this methodological problem. Our approach has been to scale down the problem size in such a way that our benchmarks exhibit an acceptable algorithmic speedup on the small machine configuration of 16 processors that we use. In Table 2 we show the speedup for the parallel section and the problem sizes we consider for our benchmarks. We will first show the results for the smaller problem sizes, then, in Section 7, we will show some limited experiments on the effects of increases in the problem sizes.

Table 2: Characteristics of the benchmarks

BENCHMARK	SPEEDUP	NUMBER OF WRITES	NUMBER OF READS	NUMBER OF ACQ/REL	DATA SET SIZE(byte)
MP3D1000	10.9	357,942	948,345	90,411	36,000
MP3D10000	14.9	1,510,524	2,561,297	411,373	360,000
WATER16	12.3	83,008	973,675	9,281	10,880
WATER288	14.9	5,114,030	71,134,138	531,709	195,840
LU32	5.7	37,386	136,454	4,043	8,192
LU200	14.9	5,663,984	11,764,532	10,995	320,000
QSORT5000	4.2	12,733	91,484	6,933	20,000
NSFFT512	15	18,432	45,232	128	8,192
JACOBI64	15	280,883	2,407,565	4,653	65,536

5. MISS CLASSIFICATION FOR THE BENCHMARKS WITH SMALL DATA SETS

The miss classifications for the six benchmarks with smaller data set sizes are displayed in Fig. 8. A general observation is that the essential miss rate goes down and the false sharing miss rate goes up as we move to larger block sizes as expected [12,22]. To understand the basic reasons, we analyzed how the data structures are accessed in our benchmark suite.

In LU32, columns go through two phases. In the first phase, they are exclusively accessed by a single processor and in the second phase, they are read-only by many. As a result, the column distribution causes cold (true) sharing misses which show up for small block sizes in Fig. 8. It is interesting to note that this component drops dramatically up until a block size of 256 bytes. The reason is that the largest columns occupy 256 bytes each. As the block size increases the cold true sharing misses turn into pure true sharing misses, an effect that was demonstrated in Fig. 5. As for false sharing, LU works on triangular matrices where columns are interleaved among processors.

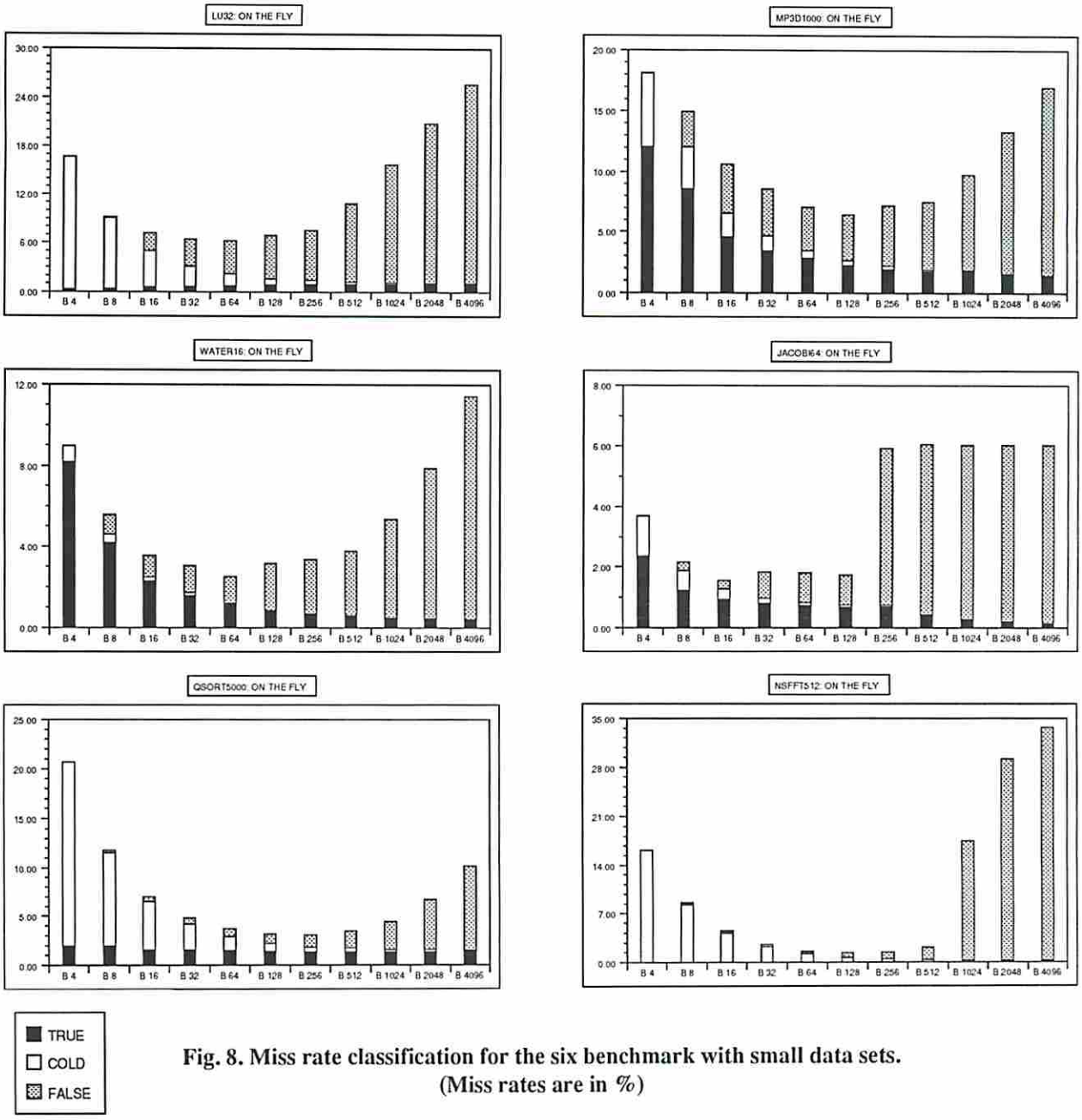

Fig. 8. Miss rate classification for the six benchmark with small data sets. (Miss rates are in %)

For the smaller columns, therefore, false sharing is expected to show up. As we can see in Fig. 8, the false sharing component is significant even in the cache range. For example, when the block size is 32 bytes, half the miss rate is caused by false sharing.

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 on 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, on the other hand, are due

to modifications of particles and space cells. Since two consecutive particle objects are allocated on different processors, false sharing starts to show for a block size of eight bytes because the object size is 36 bytes. As for the space cells, false sharing starts to show up for blocks larger than 16 bytes because the space-cell object is 48 bytes. As we move to larger block sizes, the false sharing component of course increases.

In WATER16, each processor calculates both intra-molecular and inter-molecular interactions. True sharing is caused by the 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, as we can see in Fig. 8, the true sharing miss rate component decreases rapidly until a block size of 128 bytes. As for the false sharing, it mainly results from accesses to two consecutively allocated molecules belonging to different processors. The false sharing rate goes up when the block size approaches the size of the molecule data structure, which is 680 bytes.

In JACOBI, each processor works on a submatrix of size 16x16. Since each matrix element is a double word (8 bytes) we would expect the 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 both these effects in Fig. 8. As for false sharing, we would expect to see the false sharing component from the matrix elements to show up when the block frame partly covers the data partitions of two processors. Since the size of the 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. 8. It is noteworthy to observe that false sharing arises at a block size of 8 bytes. This false sharing is due to the large number of barrier synchronizations in the program (one after each iteration) and also to 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.

In NSFFT most of the communication is done through cold misses since other processors' modified values are read once after the values have been modified. Therefore, it has mostly cold true sharing misses in the caching range with a slowly rising false sharing miss component. False sharing is due to the fact that large blocks can be allocated across array partitions and therefore a write by a neighboring processor may cause the invalidation a block of other's. There should be a jump in the number of false sharing once the block size approaches or exceeds the data partition of each processor. This block size is 1024 bytes.

QSORT as expected has a large component of cold misses which nicely decreases with the block size. The threshold for insertion sort [19] was selected at 15 data items, which means that little false sharing is observed for block sizes under 64 bytes.

In all six benchmarks, the essential miss rate (cold plus true misses) steadily goes down as the block size goes up; however, the false sharing miss component becomes larger and larger and more than offset this decrease. The curves of Fig.8 were obtained for the on-the-fly protocol in which all stores are executed and their invalidations are propagated in the order of the trace. As previously argued, the scheduling of stores and their invalidations may alter the miss rate as well as the distribution of misses. In the following, we present simulation results to demonstrate this effect. We are mostly interested in schedules which improve the miss rate, but we also show a schedule which causes more misses than the on-the-fly protocol.

6. EFFECTS OF INVALIDATION SCHEDULES

The new miss classification indicates that the shared data miss rate can theoretically be reduced to its essential component, by delaying and combining invalidations until a miss is required for the correctness of execution. We have simulated the seven schedules described in Section 3, including the worst case MAX. We report here on the simulation results both in the caching and in the paging ranges. In the caching range, we have selected to display results for block sizes of 16, 32, 64, and 128 bytes. These block sizes are the ones of the caches of the DASH multiprocessor [18](and of the SGI cluster), of the Sequent Symmetry, of the IEEE Scalable Interface standard [15] and of the IBM RS6000 workstation, respectively. In the paging range, we have selected to display results for a page of 1,024 bytes; there was very little qualitative difference between the results for pages of 1K, 2K, and 4K bytes.

6.1 Caching Range

In Fig. 9 and 10 we show the comparison between the miss rates of LU32, MP3D1000, WATER16 and JACOBI64 in the caching range. The decomposition into PTS, COLD and PFS misses is shown except for MIN (which has no false sharing), WBWI and MAX (we only display the total miss rate for these three.) The results for QSORT and NSFFT in the caching range are simply not interesting due to the very low level of false sharing.

All protocols except the write-through protocol MIN suffer from the cost of maintaining ownership. A store miss is triggered on a store to a non-owned block if any one of the words in the block has been invalidated. This importance of this effect can be understood by comparing WBWI and MIN, because the only difference between these protocols is ownership. Remarkably, Fig.9 and 10 show that the cost of ownership is very low in the caching range, for the four benchmarks. This can be explained easily: When the block size is small, the probability that more than one processors are writing in different part of the same block at the same time is very low.

The protocols relying on word invalidations have high invalidation traffic because invalidations are aimed at words instead of blocks. They also require one dirty bit per word. In the caching range, for relatively small block sizes, word invalidation is very effective and its overhead may be acceptable. For larger block sizes, it may affect performance and cost. One advantage of word invalidation is that it is applicable to systems with any memory consistency models.

Another approach to eliminating false sharing relies on synchronizations and on weakly ordered memory models to delay and combine invalidations as much as possible. Invalidations can be delayed when they are sent or when they are received. Delaying the sending of invalidations is similar to executing stores later in each processor. The order of the trace is therefore changed. This may change the number of essential misses (see Fig. 6) but we have not observed any of that in all our simulation runs. For all the delayed protocols, the essential miss component was always the same in the case of each benchmark. Stores can be delayed until the execution of the next synchronization or even, more aggressively, until the next release. If multiple stores for the same block are delayed and are *combined* in one single invalidation at the sending end, only one invalidation is sent instead of one invalidations for each store. This can drastically reduce the false sharing miss rate when another processors is also writing in a different part of a block at the same time. However, this situation does not seem to occur very frequently in the caching range because the blocks are too small. In general, our simulations show that pure send-delayed proto-

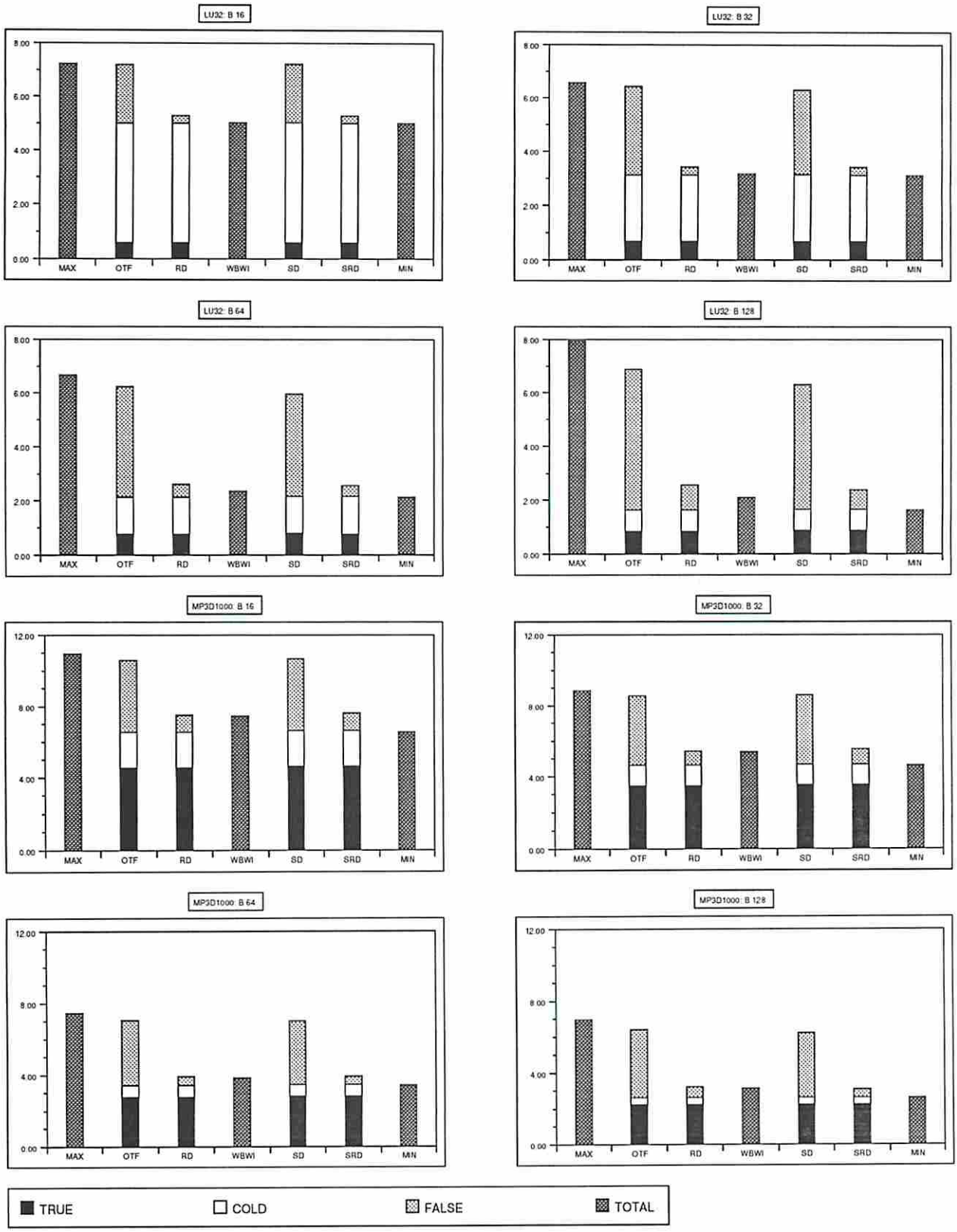

Fig.9 Effect of invalidation scheduling on the miss rate (%). LU32 and MP3D1000 Block size: 16,32,64 and 128 bytes (caching range)

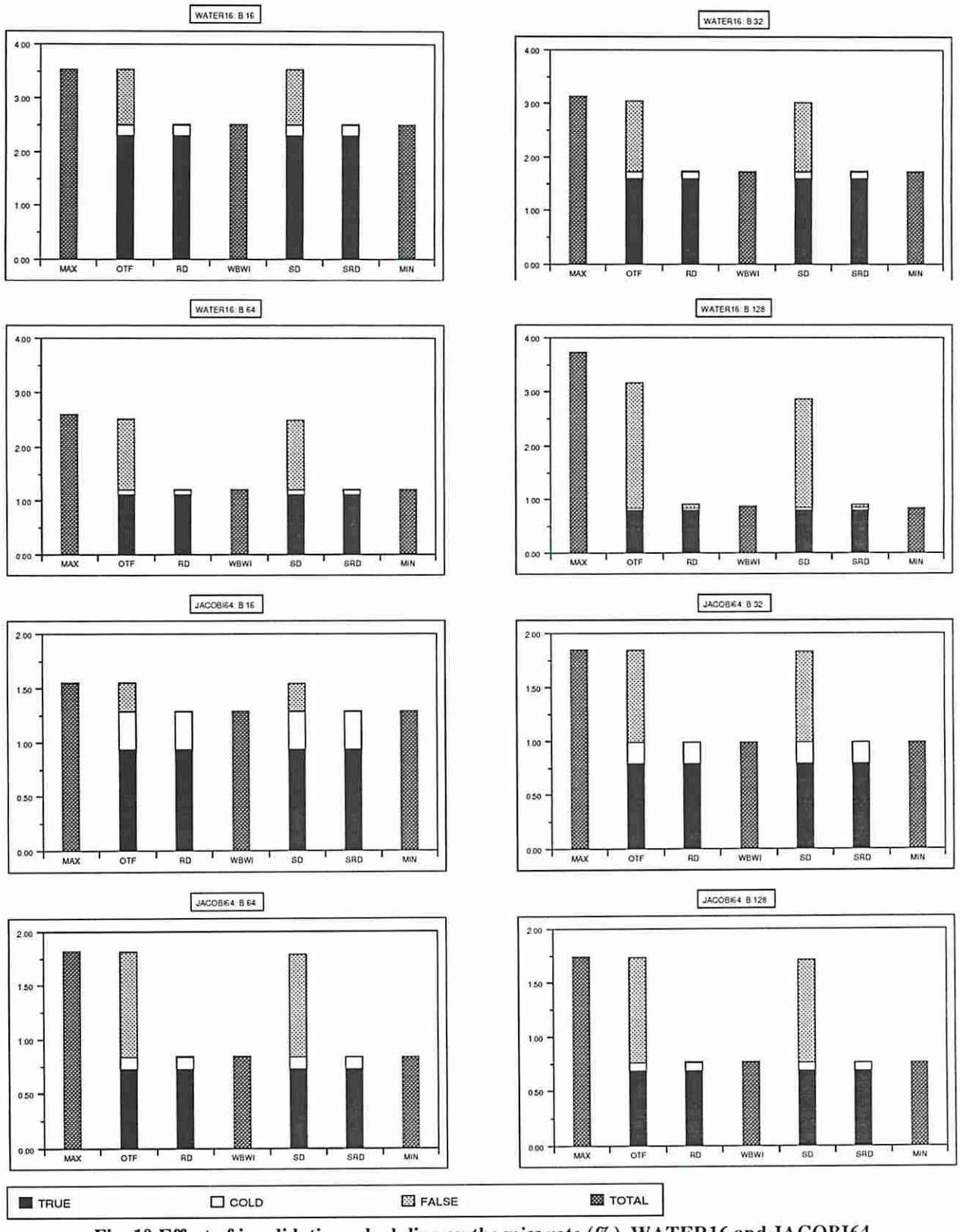

Fig. 10 Effect of invalidation scheduling on the miss rate (%). WATER16 and JACOBI64. Block size: 16,32,64 and 128 bytes (caching range)

cols are not very effective in the sense that their miss rate is still far from the optimum, essential miss rate. The overhead of *send delayed* protocols is to buffer locally partially modified copies of a block while the stores are being delayed; complexity is added in the handling of misses because partially modified block copies have to be merged. It can be argued that such buffering will have to take place eventually anyway in order to tolerate large invalidation latencies.

In *receive delayed* protocols, received invalidations for the same block are delayed and *combined* until a stale value might be accessed locally, as in the word invalidate protocol. To detect this possibility, we rely on synchronization accesses. Invalidations must be executed at the next synchronization point or even at the next acquire. RD protocols cause more false sharing than word invalidate protocol because the synchronization access causing the invalidation can be very far from the access to the stale word. It could even be that the synchronization access causing the invalidation of a block is unrelated to and does not protect the stale word. Received delayed protocols can be implemented with a physical buffer or with a stale bit per block frame in the cache. The stale bit is set when the invalidation is received, and then, at the following acquire, the stale bit is ORed with the Invalid bit [10]. The invalidation rate of RD is lower than that of WBWI because invalidations target entire blocks instead of words. By comparing the miss rate for the RD and WBWI protocols, in Fig. 9 and 10, we can estimate the effect of the distance between the acquire causing the invalidation and the stale word, since the detection of stale words is the only difference. It appears that this effect is negligeable.

Overall, in the caching range, all delayed protocols are always effective and have about the same number of misses except for the SD protocol, which is mostly not effective. The miss rates for the delayed protocols (except for SD) are also close to the optimum, which is the miss rate of MIN, the essential miss rate in the trace. Therefore further improvements are impossible. We want to stress here the importance of a correct classification of misses in order to understand how good a protocol is or a compiler solution is at removing useless misses. For example, for LU32, using Eggers's classification, the number of essential misses is only about half the correct number, for a block size of 32 bytes. This measure would have led us to believe that significant additional reductions of the miss rate were still possible.

6.2 Paging Range

The comparison between the schedules is shown in Fig. 11, for a block size of 1024 bytes. In the paging range, the miss rates of both WBWI and RD are still very high, as compared to the miss rate of MIN. This is because of ownership. Since the block is large, the chance that several processors write in the same block at the same time is high; therefore the cost of maintaining ownership is much higher in the paging range. This effect added to the effect of store combining (for which there is more opportunities in the case of a page-size block) makes the SD protocol much more attractive than in the caching range. Since WBWI requires to maintain one bit per word in the page as well as to send one invalidation for each word of the page, RD is definitely preferable to WBWI since it exhibits almost the same miss rate. The combination of send delayed and receive delayed is very effective, except in the case of quicksort. In this case, the combining due to the delayed send is not sufficient to offset the cost of maintaining ownership. One problem with the quicksort is the large amount of synchronizations (which in turns limit the amount of store combining). The reason is that the modula-P compiler assumes a strongly ordered memory system and uses flags in some cases to synchronize threads. To avoid data races, we had to frame each

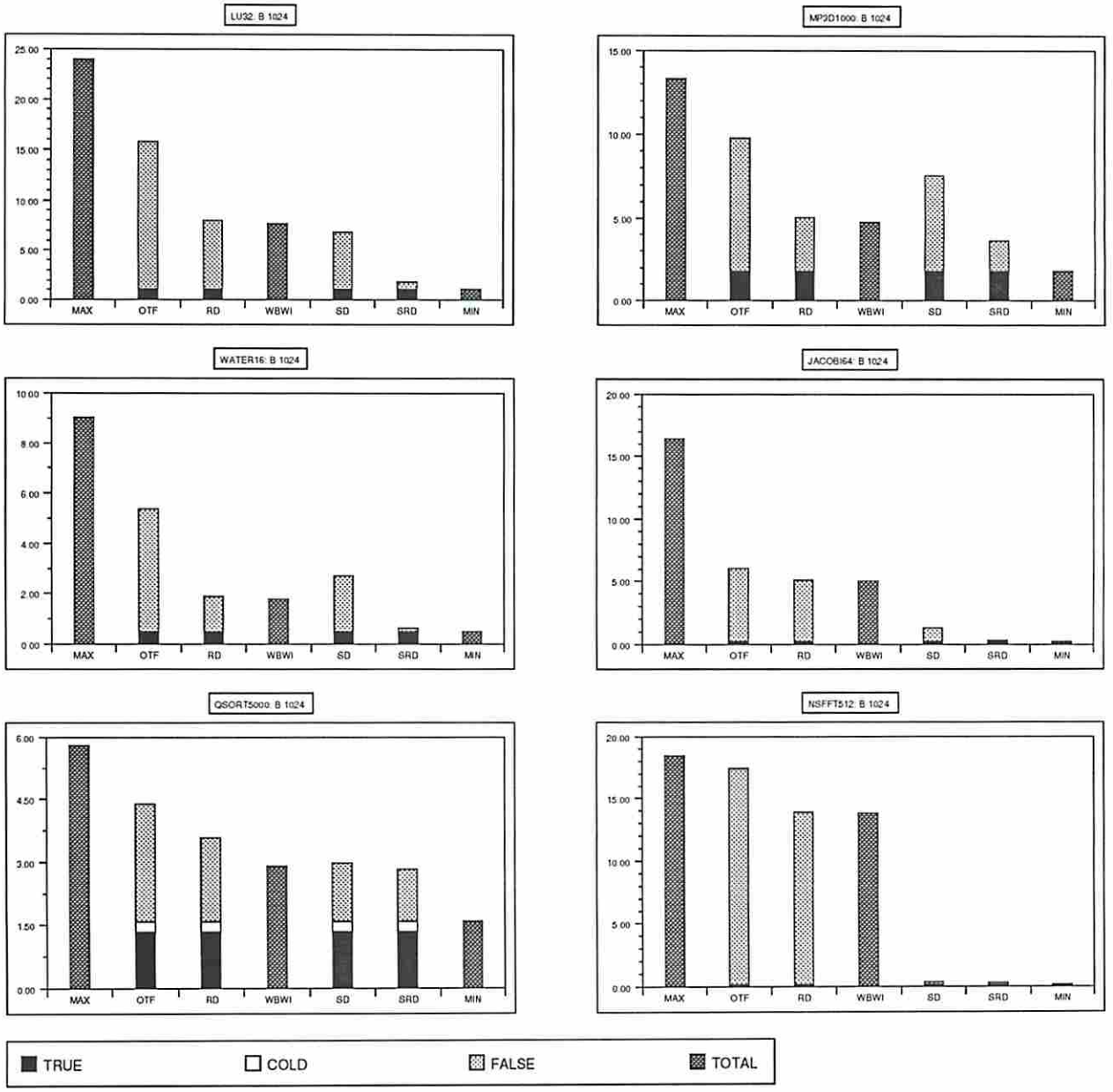

Fig. 11 Effect of invalidation scheduling on the miss rate (%). Benchmarks with small data sets. Block size: 1024 bytes (paging range)

flag access with an acquire and a release. We are currently improving the benchmarks.

In general, contrary to the caching range, the result for the paging range shows that further improvement is possible because the best protocol (SRD) does not always reach the essential miss rate of the trace, in the cases of LU, MP3D, and QSORT. 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 [9], or even systems in which coherence is maintained on words.

6.3 Worst case schedule

The MAX scheduling of invalidations always yields more misses than any other schedule,

including the on-the-fly protocol. In the caching range, the worst case schedule gave a miss rate almost equal to the on-the-fly miss rate: 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. So it appears that scheduling stores according to the release consistency model is not likely to be a problem, in the caching range. As the block size increases, however, the miss rate may become significantly worse than the on the fly miss rate. Delaying invalidations at the receiving end should also help avoid worse case situations like in the MAX schedule.

7. EFFECTS OF DATA SET SIZES

We were able to run some simulations for the larger data set sizes namely for LU200, MP3D10000 and WATER288. Fig. 12 shows that the effect of false sharing has moved to higher block sizes. A lot of false sharing remains in the paging range. We discuss the effect of increasing the data set size below.

To begin with, when the data set size goes up, the size of the partition allocated to each processor goes up too. If the object size grows with the data set size, false sharing shows up at larger block sizes. An example of this effect is the columns in LU200. In LU, the unit of partition is a column. The largest column is 1600 bytes. As expected, the false sharing component drastically goes up at a block size of 2048 bytes instead of at 256 bytes that happened for LU32. In WATER288, we see the same shift of false sharing to larger block sizes. The explanation is that each processor now works on 18 molecules instead of 1, and these molecules are consecutively stored in the address space. Although the false sharing component has decreased in MP3D, it is still significant in the cache range because two consecutive particle objects are assigned to different processors.

We are note showing the curves for the caching range. The effects of invalidation sched-

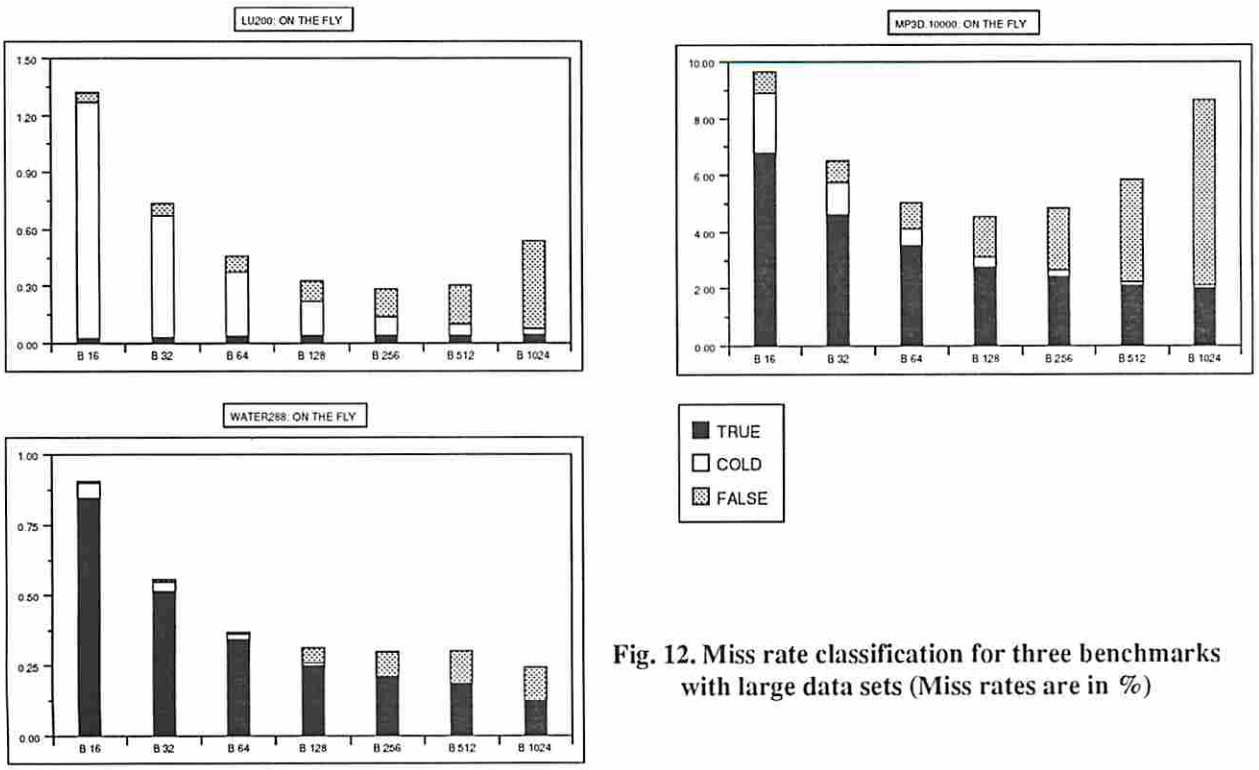

Fig. 12. Miss rate classification for three benchmarks with large data sets (Miss rates are in %)

ules are similar to the case of small data sets, but these effects are much reduced since the difference between the on-the-fly miss rate and the essential miss rate is always less than 30%. The results are shown in the paging range in Fig. 13. The trends are the same as for Fig. 11. Note however the very large miss rate yielded by MAX in the case of LU.

8. CONCLUSIONS

In this paper, we have introduced a classification of multiprocessor cache misses based on the fundamental concept of inter processor 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. Therefore the number of essential misses gives us a minimum on the possible miss rate of a trace. The rest of the misses are false sharing misses. False sharing misses are useless misses: after a false sharing miss, the processor does not read any new value defined since the last essential miss during the entire lifetime of a block in a cache. Therefore, the trace would execute correctly even if the false sharing misses (or alternatively the invalidations leading to these misses) were not executed in the cache.

We have shown that previous classifications tend to overestimate the amount of false sharing by applying the various classifications to a set of six benchmarks. For the small data set sizes (fine granularity of parallelism) and four of the benchmarks there was a significant component of false sharing in the miss rate even for relatively small 16-byte block sizes. The false sharing component usually shoots up in the paging range. In general, protocols to reduce false sharing will be

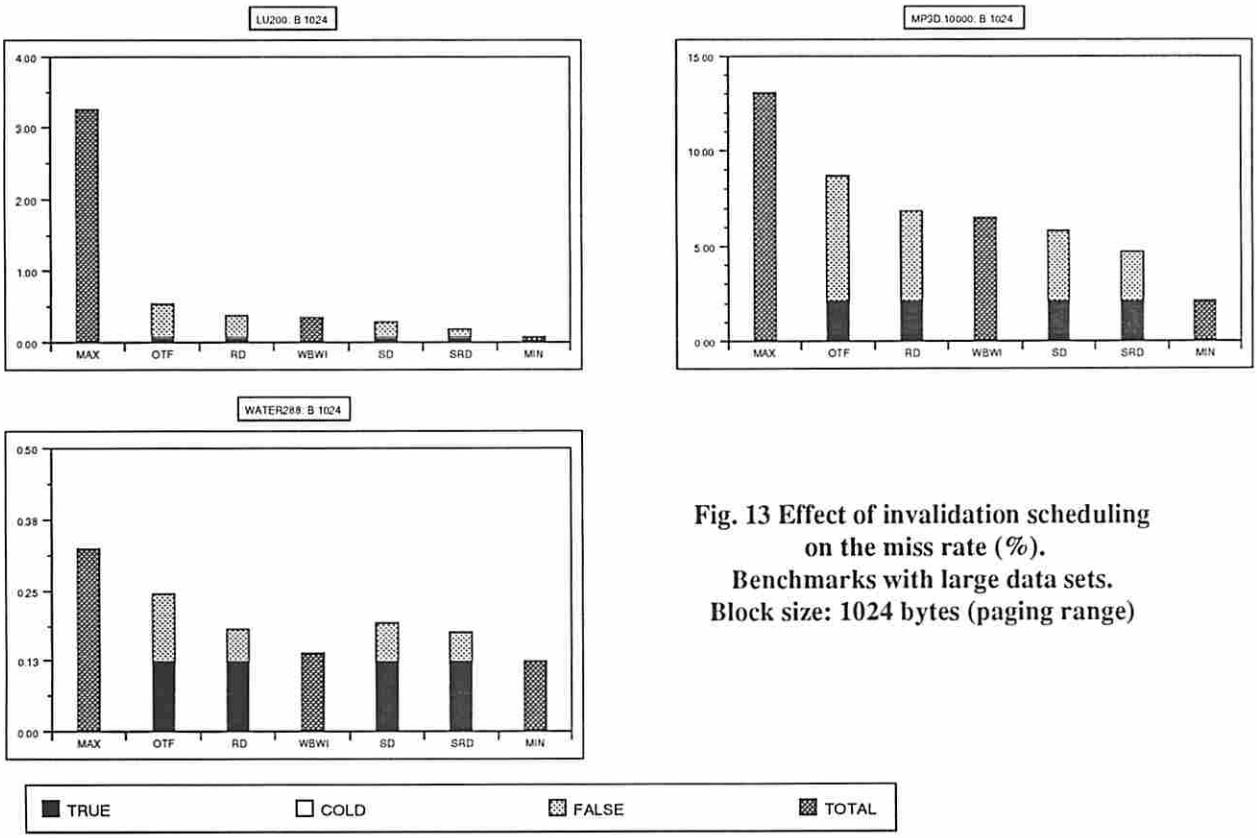

Fig. 13 Effect of invalidation scheduling on the miss rate (%). Benchmarks with large data sets. Block size: 1024 bytes (paging range)

needed in virtual shared memory systems. They are also needed in cache-based systems, for programs with relatively fine granularity of parallelism.

We have also simulated several approaches to effectively detect and eliminate useless misses by dynamically delaying and combining invalidations. In the caching range, all these techniques were very effective, except for the pure send-delayed protocol; in this range there is very little room for improvement. Considering hardware complexity, write-back word invalidate protocol may be preferable for small block sizes whereas a pure receive-delayed protocol is probably preferable for large block sizes. The receive delayed protocol is only applicable to systems with weak ordering or release consistency relying on synchronizations to enforce consistency.

In the paging range the techniques addressed in this paper are particularly needed. Delaying and combining invalidations at both the sending and receiving end is useful but not sufficient to remove all useless misses. One reason is the need to maintain ownership. Our results show that further improvement is still possible if one can deal with the problem of ownership.

Finally, we have not displayed the amount of miss traffic generated by these protocols. The miss traffic is the miss rate multiplied by the block size. The protocols with reduced miss rates also have reduced miss traffic. However, even with the miss reduction obtained by these protocols, the miss traffic is very high for large block sizes, especially in the paging range. At this level of traffic, delayed write-broadcast or delayed competitive protocols, which can reduce the number of essential misses, may become attractive. Besides the scheduling of store, loads can also be scheduled to hide the latencies of load misses and it would be very interesting to understand how prefetching and lockup-free caches affect the false sharing and essential miss rates. Our future research will concentrate on these possibilities.

9. REFERENCES

[1] Bennett, J.K., Carter, J.B., and Zwaenepoel, W. "Adaptive Software Cache Management for Distributed Shared Memory Architectures," *Proc. of the 17th Annual Int. Symposium on Computer Architecture*, June 1990, pp. 125-134.

[2] Bitar, P., "A Critique of Trace-Driven Simulation for Shared-Memory Multiprocessors", in *Cache and Interconnect Architectures in Multiprocessors*, M. Dubois and S. Thakkar ed., Kluwer Academic Publishers, June 1990.

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

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

[5] Brorsson, M., Dahlgren, F., Nilsson, H., and Stenström, P.,"The CacheMire Test Bench --- A Flexible and Effective Approach for Simulation of Multiprocessors". Technical Report, Dept. of Computer Engineering, Lund University, Sweden, September 1992.

[6] Bray, K.B. and Flynn, M.J., "Write Caches as an Alternative to Write Buffers," Technical report No. CSL-TR-91-470, Stanford University, Apr. 1991.

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

[8] Digital Signal Processing Committee, editor. Programs for Digital Signal Processing. IEEE

Press, 1979.

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

[10] 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.

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

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

[13] Francis, R., "The PRISM Multiprocessor Simulator," High Performance Computation Project, Division of Information Technology, C.S.I.R.O., 723 Swanston St., Carlton 3053, Australia.

[14] Francis, R., "ModulaP Language Reference Manual," High Performance Computation Project, Division of Information Technlology, C.S.I.R.O., 723 Swanston St., Carlton 3053, Australia.

[15] Gustavson, D. B., "The Scalable Coherent Interface and Related Standards Projects", *IEEE Micro,* Vol. 12, No. 1, pp. 10-22, February 1992.

[16] Hill, M.D., "Aspects of Cache Memory and Instruction Buffer Performance," Tech. Rep. UCB 87/381, University of California, Berkeley, November 1987.

[17] Keleher,P., Cox,A.L., and Zwaenepoel,W.,"Lazy Release Consistency for Software Distributed Shared Memory," *Proc. of the 19th Ann. Int. Symp. on Computer Architecture*, May 1991, pp. 13-21.

[18] 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*, June 1990.

[19] Sedgewick, R., "Quicksort", New York: Garland Publishing, Inc., 1980.

[20] 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.

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

[22] 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.*, Aug 1990, pp. 266-270. Also published as "Measurement, Analysis, and Improvement of the Cache Behavior of Shared Data in Cache Coherent Multiprocessors" Technical report CSL-TR-90-412, Stanford University, Feb. 1990.

[23] Weber, W-D, and Gupta, A.,"Analysis of Cache Invalidation Patterns in Multiprocessors," *Proc. of the 3rd Int. Conf. on Arch. Support for Prog. Languages and Systems(ASPLOS)*, pp.243-256, Apr 1989.

[24] Young, D., "Iterative Solution of Large Linear Systems", Academic Press: New York, 1971.