12

Algorithms for Categorizing Multiprocessor Communication under Invalidate and Update-Based Coherence Protocols

R. Bianchini and L.I. Kontothanassis

Technical Report 533
September 1994

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Algorithms for Categorizing Multiprocessor Communication Under Invalidate and Update-Based Coherence Protocols

Ricardo Bianchini and Leonidas Kontothanassis

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226

{ricardo,kthanasi}@cs.rochester.edu

Technical Report 533

September 1994

Abstract

In this paper, we present algorithms that characterize the main sources of communication generated by parallel applications under both invalidate and update-based cache coherence protocols. The algorithms provide insight into the reference and sharing patterns of parallel programs and into the amount of useless traffic entailed by each coherence protocol. Under an invalidate-based protocol, our algorithms classify the data traffic caused by the different types of cache misses. Under an update-based protocol, our algorithms not only categorize the data traffic, but also classify update transactions with respect to the sharing patterns that caused them. Although our algorithms deal with numerous hardware features such as finite-sized caches and coalescing write buffers, our categorization is widely applicable and can be easily simplified for use in less detailed environments. Our work extends previous categorizations of cache misses in write-invalidate protocols, while introducing a new categorization of the coherence traffic in update-based protocols.

This research was supported in part by Brazilian CAPES and NUTES/UFRJ fellowships, and by NSF Institutional Infrastructure grant no. CDA-8822724 and ONR research grant no. N00014-92-J-1801 (in conjunction with the DARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930).

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE September 1994	3. REPORT TYPE AND DATES COVERED technical report

4. TITLE AND SUBTITLE Algorithms for Categorizing Multiprocessor Communication ...	5. FUNDING NUMBERS N00014-92-J-1801, ARPA Order 8930
6. AUTHOR(S) R. Bianchini and L.I. Kontothanassis	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES Computer Science Dept. 734 Computer Studies Bldg. University of Rochester Rochester NY 14627-0226	8. PERFORMING ORGANIZATION
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESSES(ES) Office of Naval Research / Information Systems / Arlington VA 22217 ARPA / 3701 N. Fairfax Drive / Arlington VA 22203	10. SPONSORING / MONITORING AGENCY REPORT NUMBER TR 533
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION / AVAILABILITY STATEMENT Distribution of this document is unlimited.	12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 words) (see title page)	
14. SUBJECT TERMS multiprocessor communication; cache coherence protocols; write invalidate; write update; scalable multiprocessors	15. NUMBER OF PAGES 21 pages 16. PRICE CODE free to sponsors; else $2.00

17. SECURITY CLASSIFICATION OF REPORT unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT unclassified	20. LIMITATION OF ABSTRACT UL

1 Introduction

Large-scale shared-memory multiprocessors provide the computational power and the ease of programming needed to tackle some of the larger problems of science and engineering today. Such machines use hardware caches to reduce the average cost of a data access by storing data close to processors that need it.

Although processor caches are very successful at reducing the amount of communication found in parallel applications, the remaining traffic may still be significant. The most important source of communication exhibited by programs depends on the coherence protocol used. The cache coherence protocol determines how data is moved between caches in the machine, ensuring that data is frequently found in the local cache, while preventing processors from using stale data.

There are two common classes of coherence protocol used in shared-memory machines: write-update protocols (WU) [3, 11, 14] and write-invalidate protocols (WI) [6, 10, 12]. Under a WU protocol, each time a processor writes shared data, the coherence protocol broadcasts the new value to every other processor caching that data. Under a WI protocol, a write to a shared cache block causes the coherence protocol to send invalidation messages to all processors caching copies of the block.

Under both coherence schemes, data traffic (caused by cache misses and writebacks) is responsible for a large fraction of the total communication, particularly when cache blocks are relatively large [7]. Under an update-based protocol, the coherence traffic also becomes a significant part of the total traffic, since all write accesses to shared data require communication. Although some of the communication entailed by these types of coherence protocol cannot be avoided, it is often the case that a large fraction of the communication traffic is simply useless.

Both WI and WU schemes have the potential to introduce useless communication due to a mismatch between the hardware unit of coherence and the data structures manipulated by the user program. As an example of such a mismatch, consider a situation where two processors read and write *different* portions of a multi-word cache block. In this case, the coherence protocol has to maintain the copies of the block consistent, even though each processor will never use the other processor's changes. Regardless of the protocol, this type of communication is unnecessary from a correctness standpoint. [1]

Relatively small caches can also cause unnecessary communication, due to replacement misses in the absence of good temporal locality. An excessive amount of unnecessary traffic may significantly slow programs down by increasing the number and duration of processor stalls. After detected, useless communication can be eliminated either by using efficient coherence protocols, clever compilers, or program restructuring techniques.

Our work is concerned with detecting and categorizing useless communication under WI and WU coherence protocols. We focus on classifying the major sources of communication exhibited by programs running under these protocols. We extend a well-known algorithm for classifying cache misses, in order to account for finite-sized caches under invalidate-based protocols. In addition, we introduce an algorithm that accurately characterizes update transactions under a write-update protocol (extensions of this algorithm consider coalescing write buffers and competitive update strategies).

[1]The term *false sharing* has been used to describe this type of interference in write-invalidate protocols [1, 4, 5, 16].

Finally, we present simulation results that characterize the communication encountered in various well-known parallel programs under invalidate and update-based protocols.

The remainder of this paper is organized as follows. Section 2 presents the algorithms for the classification of cache misses and update transactions. Section 3 describes our simulation infrastructure and application workload. Section 4 presents the results of our categorization for our application suite. Section 5 relates our approach to similar work found in the literature. In section 6 we summarize our findings and conclude the paper.

2 Algorithms

In this section we present the algorithms for classifying the major sources of communication under both WI and WU coherence protocols. Under the invalidate-based protocol, the major source of communication corresponds to data (cache block) transfers caused by cache misses and writebacks. Since the data traffic caused by writebacks is usually overwhelmed by the miss traffic, we limit ourselves to classifying cache misses. Under the WU protocol, we not only classify cache misses, but also categorize update transactions. Under an update-based protocol, cache misses and updates (and their associated acknowledgements) account for the vast majority of the communication traffic.

We assume a simulator structure that has separate routines for handling read misses, read hits, write misses, write hits and invalidations. The code we present is an addition to the code implementing those routines and handles only the classification of the major sources of communication traffic.

2.1 Data Traffic Under a WI Protocol

Our algorithm for invalidate-based coherence is a simple extension of the one presented in [4]. We categorize cache misses in terms of the reference and sharing behavior causing them. We identify four basic categories for misses:

- **Cold start misses.** A cold start miss happens on the first reference to a block by a processor.
- **True sharing misses.** A true sharing miss happens when a processor references a word belonging in a block it had previously cached but has been invalidated, due to a write by some other processor to the same word.
- **False sharing misses.** A false sharing miss occurs in roughly the same circumstances as a true sharing miss, except that the word written by the other processor is not the same as the word missed on.
- **Eviction misses.** An eviction (replacement) miss happens when a processor replaces one of its cache blocks with another one mapping to the same cache line and later needs to reload the block replaced.

Cold start and true sharing misses are necessary for the correct execution of the program so they can be thought of as useful (essential) misses, while false sharing and eviction misses represent shortcomings of the architecture and/or the program and thus are considered useless misses.

```
void
read_hit_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  if (Comm[proc_id, word]) {
    Essential[proc_id, block_id] = True;
    foreach wrd in block_id
      Comm[proc_id, wrd] = False;
  }
}
```

```
void
read_miss_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  if (Infinite[proc_id, block_id]) {
    /* Should have been in the cache */
    M_evict++;
    Classified[proc_id, block_id] = True;
  } else {
    /* Don't know if miss is useful yet */
    Essential[proc_id, block_id] = False;
    Classified[proc_id, block_id] = False;
    read_hit_class(proc_id, block_id, word);
  }
}
```

```
void
write_hit_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  if (!Dirty[proc_id, block_id])
    M_excl++;
  else
    read_hit_class(proc_id, block_id, word);
  foreach proc not sharing block_id
    Comm[proc, word] = True;
}
```

```
void
write_miss_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  read_miss_class(proc_id, block_id, word);
  foreach proc not sharing block_id
    Comm[proc, word] = True;
}
```

Figure 1: Algorithm for the classification of cache misses under a WI protocol.

```
void
invalidate_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  Infinite[proc_id, block_id] = False;
  if (!Classified[proc_id, block_id])
    classify(proc_id, block_id);
  Comm[proc_id, word] = True;
}

void
end_program_class()
{
  foreach proc
    foreach block
      if (!Classified[proc, block])
        classify(proc, block);
}

void
complete_miss_request_class(proc_id, block_id)
{
  Infinite[proc_id, block_id] = True;
}
```

```
void
replacement_class(proc_id, block_id)
int proc_id, block_id;
{
  if (!Classified[proc_id, block_id])
    classify(proc_id, block_id);
}

void
classify(proc_id, block_id)
int proc_id, block_id;
{
  if (Present[proc_id, block_id]) {
    if (!Cold[proc_id, block_id]) {
      M_cold ++;
      Cold[proc_id, block_id] = True;
    } else
      if (Essential[proc_id, block_id])
        M_true ++;
      else
        M_false ++;
    Classified[proc_id, block_id] = True;
  }
}
```

Figure 2: Algorithm for the classification of cache misses under a WI protocol (Cont).

Figures 1 and 2 present the algorithm for classifying cache misses under a WI protocol.[2] Misses are classified at the end of a block's lifetime in the cache, an event that occurs as a result of an invalidation, a replacement, or the termination of the program. An exception to this rule is eviction misses (and exclusive requests) which are classified the moment the block is brought in the cache, since status of the block cannot change until the end of its lifetime.

The main data structures used in this algorithm are six two-dimensional arrays indexed by processor identification numbers and cache block numbers, and a matrix (**Comm**) indexed by processor numbers and word addresses. The functionality of each data structure is the following:

Comm. This bit array saves information about writes to each of the words in the system. When a processor writes a word, all remaining processors have their corresponding **Comm** bits set for that word. When a processor accesses a word, it checks to see if its **Comm** bit is set. If this is the case and the miss that brought the block into the cache is not a cold miss, there

[2]Note that our algorithm includes a fifth category, exclusive request transactions. An exclusive request operation (caused by a write to a read-shared block already cached by the writing processor) is not strictly a cache miss, although the processor may have to stall until it receives ownership of the block.

was useful communication between processors and the corresponding miss is marked as a true sharing miss.

Cold. This bit array is used to classify cold start misses. A **Cold** bit is set when the first miss by a processor on a certain block is detected by the algorithm.

Essential. When a miss occurs it is marked as non-essential. Future references to the cache block may change this characterization to essential, if the processor accesses a word whose **Comm** bit is set. When a cache block is thrown out of the cache by some processor, the corresponding **Essential** bit contains the information on whether the miss that originally brought the block into the processor's cache was useful.

Classified. This array is used to make sure that a miss classified as an eviction miss at the beginning of a block's lifetime does not get reclassified at the end of a block's lifetime.

Infinite. This array represents the caches' contents assuming that caches are infinite. A miss on a block present in the processor's cache according to the **Infinite** array determines a replacement miss.

Present and **Dirty.** These arrays represent the present and dirty bits associated with each block in the caches. These arrays must already exist in any simulation of caches.

2.2 Data and Update Traffic Under a WU protocol

Our algorithm for classifying the dominant sources of communication traffic under write-update categorizes both cache misses and update transactions. Classifying the update traffic posed the difficult problem of defining meaningful categories for updates. We believe that our categorization, while by no means unique, captures the important characteristics of update traffic in an intuitive manner.

Since, in a pure write-update protocol, cache blocks are never invalidated, cache misses can be only of two kinds: cold start and eviction misses. We have identified four categories of update transactions:

- **True sharing updates.** The receiving processor references the word modified by the update message before another update message to the same word is received. This type of update transaction is termed useful, since it is necessary for the correctness of the program.

- **False sharing updates.** The receiving processor does not reference the word that is modified by the update message before it is overwritten by a subsequent update, but references some other word in the same cache block. This class of updates is part of the larger subclass termed useless updates.

- **Proliferation updates.** The receiving processor does not reference the word modified by the update message before it is overwritten by a subsequent update, and it does not reference any other word in that cache block either. Proliferation updates are also useless in terms of the correctness of the program.

```
void
read_hit_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  Updused[proc_id, word] = True;
  foreach wrd in block_id
    Refd[proc_id, wrd] = True;
}
```

```
void
read_miss_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  if (!Cold[proc_id, block_id]) {
    Cold[proc_id, block_id] = True;
    M_cold ++;
  } else
    M_evict ++;
  read_hit_class(proc_id, block_id, word);
}
```

```
void
write_hit_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  read_hit_class(proc_id, block_id, word);
}
```

```
void
write_miss_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  read_miss_class(proc_id, block_id, word);
}
```

```
void
recv_upd_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  if (!First[proc_id, word])
    First[proc_id, word] = True;
  else {
    if (Updused[proc_id, word])
      U_true ++;
    else if (Refd[proc_id, word])
      U_false ++;
    else if (end_of_program)
      U_end ++;
    else
      U_prolif ++;
  }
  Updused[proc_id, word] = False;
  Refd[proc_id, word] = False;
}
```

```
void
end_program_class()
{
  end_of_program = True;
  foreach proc
    foreach block
      foreach word
        recv_upd_class(proc, block, word);
}
```

Figure 3: Algorithm for the classification of update transactions under a WU protocol.

- **End updates.** An end update is a proliferation update happening at the end of the program. This class of updates belongs in the useless updates subclass as well.

Figure 3 presents the algorithm for classifying update transactions under a WU protocol. In this algorithm, data-related communication consists solely of cold start and eviction misses. These misses are classified at the moment they happen, since their status cannot change afterwards. Update messages are classified at the end of an update's lifetime, which happens when it is overwritten by another update to the same word or when the program ends.

The algorithm is fairly straightforward with the exception of the false sharing classification. False sharing is classified by marking all words of a block as referenced when any word in the block is accessed. Information is kept on a per word basis to allow for successive (useless) updates to the same word in a block to be classified as proliferation instead of false sharing updates. Thus, our algorithm classifies useless updates as proliferation, unless *active* false sharing is detected or the application terminates execution.

The most important data structures used for the classification of update transactions are three two-dimensional arrays of bit flags indexed by the processor identification number and the word referenced. The functionality of each data structure is the following:

Updused. When a processor reads or writes a word in a cache block, the corresponding entry of Updused is set, signifying that the word has been used. Upon receipt of an update for a word, the algorithm checks the corresponding entry in this array. If it is set then the previous update is classified as a true sharing update.

Refd. When a processor accesses a cache block all words in the block are marked as referenced by setting the corresponding entries of Refd. When an update transaction is found not to be a true sharing one, this array is examined to decide between the remaining categories. If the corresponding bit in the array is set, it implies that some other word in the block has been referenced and, therefore, that the block is undergoing active false sharing. If the Refd bit is not set, no words were referenced between the two updates and the cache line is not being used. In this case, the previous update belongs in the proliferation category.

First. This array allows us to postpone classification of update transactions until there have been at least two updates to the same word.

2.3 Update Transaction Categorization with Coalescing Write Buffers

While the above two algorithms provide the necessary framework for classification schemes under both WU and WI protocols, they do not cover all possible coherence protocol and architectural variations. However, we believe that they provide the basis for classification algorithms for any hardware platform and coherence scheme. In the remainder of this section, we will show how we can adapt the algorithms presented above to classify the major sources of communication for hardware that coalesces multiple updates into a single message and for protocols that use hybrid invalidate-update mechanisms.

Coalescing [8] is a technique that merges writes to the same cache line and only sends them out when the number of entries present in the coalescing buffer exceeds a certain value. The problem

```
void
receive_message_class(proc_id, block_id, words, num_written)
int proc_id, block_id, num_written;
int *words;
{
  U_true  = 0;
  U_false = 0;
  for (i = 0; i < num_written; i ++) {
    if (Updused[proc_id, words[i]])
      U_true ++;
    else if (Refd[proc_id, words[i]])
      U_false ++;
    Updused[proc_id, words[i]] = False;
    Refd[proc_id, words[i]] = False;
  }
  if (U_true > 0)
    M_true ++;
  else if (U_false > 0)
    M_false ++;
  else if (end_of_program)
    M_end ++;
  else
    M_prolif ++;
}
```

Figure 4: Algorithm for the classification of update transactions under WU with coalescing.

introduced with coalescing is that updates are delivered in groups so the messages seen by the communication media are decoupled from the specific updates sent by the processors. If our goal is to classify communication (i.e. the messages sent by processors) the algorithm of figure 3 is not sufficient. It is however straightforward to extend the algorithm to account for coalescing. First, we need to extend the definitions of useful and useless updates to apply to a collection of updated words (those included in a message) as opposed to individual words. The extended definitions are as follows:

- **True sharing message.** At least one of the updates included in the message is a true sharing update.
- **False sharing message.** None of the updates included in the message is a true sharing update and at least one of the updates included in the message is a false sharing update.
- **Proliferation message.** All of the updates in the message are proliferation updates.
- **End messages.** Proliferation messages at the end of the program are classified separately as end messages.

The algorithm in figure 4 presents the routine that classifies messages in the presence of a coalescing write buffer. It is is easy to maintain the individual update statistics as well by adding

```
void
drop_block_class(proc_id, block_id)
int proc_id, block_id;
{
  Dropped[proc_id, block_id] = True;
}
```

```
void
read_miss_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  if (!Cold[proc_id, block_id]) {
    Cold[proc_id, block_id] = True;
    M_cold ++;
  } else
    if (Dropped[proc_id, block_id]) {
      M_dropped ++;
      Dropped[proc_id, block_id] = False;
    } else
      M_evict ++;
  read_hit_class(proc_id, block_id, word);
}
```

Figure 5: Algorithm for the classification of data and update traffic under a hybrid protocol.

the `U_true` and `U_false` partial counts to the true sharing and false sharing update counts before returning from the routine. The difference between these two numbers and the count argument passed in should be added either to the end updates or the proliferation updates category depending on the end of program status flag. As another alternative, the individual update statistics can be maintained by calling routine receive_upd (see figure 3) for each of the separate updates received in a coalesced update message.

2.4 Data and Update Traffic Categorization for Competitive Protocols

Combining update and invalidate protocols is only slightly more complicated from the classification point of view. In this section, we present an algorithm for classifying the data and update traffic of a simple hybrid WU+WI protocol. The strategy is inspired by the coherency protocols of the bus-based multiprocessors using the DEC AXP21064 [3]. In these machines, each node makes a local decision as to whether to invalidate or update a cache block, when it sees an update transaction on the bus. The decision depends on the presence of the block in the primary cache. (The contents of the secondary cache are a superset of the contents of the primary cache.) When the block is present in the primary cache, the cache controller updates the copy in the secondary cache and invalidates the copy in the primary cache. If the block is updated again before any reference by the processor, the cache controller invalidates the copy in the secondary cache. Thus, after at most two update transactions, an unused cache block is invalidated from the processing node. It may be desirable to change the threshold for invalidating cache blocks, depending on the architecture and sharing pattern exhibited by parallel applications.

Figure 5 presents the necessary modification to the **read_miss_class** routine (refer to figure 3) and an additional routine to be called when a block is voluntarily dropped from the cache. The bit array **Dropped** records the fact that the corresponding block has been dropped from the processor's cache. All other routines and variables in our algorithm for a WU protocol (with or without coalescing write buffers) can be used without modification for this type of hybrid protocol.

This categorization is a very simple extension of the one for a pure WU protocol. It is possible, however, to further categorize cache misses resulting from dropped blocks into true and false sharing subcategories. Such an extension of our algorithm would require a data structure analogous to the **Comm** array used for classifying cache misses under a WI protocol.

3 Methodology

This section presents our simulation infrastructure and our application workload.

3.1 Multiprocessor Simulation

We use an on-line, execution-driven simulator that exploits a mixture of interpretation and native execution to simulate unmodified MIPS R3000 object code efficiently. We simulate events at the level of processor cycles; all simulation parameters and results are expressed in terms of processor cycles. Our simulations implement all the major components of a parallel computing system: processors, caches, write buffers, the interconnection network, local memories (including their buses), and directories.

We simulate a scalable direct-connected multiprocessor with 32 nodes. Each node in the simulated machine contains a single processor, cache memory, local memory, directory memory, and a network interface. A pipelined memory bus connects these components. The interconnection network is a bi-directional wormhole-routed mesh, with dimension-ordered routing.

Each processor has a 64 KB direct-mapped data cache. We present results for three different cache block sizes: 16, 64, and 256 bytes. Our WI protocol keeps caches coherent using an implementation of the DASH protocol with release consistency [10]. In our WU implementation a processor writes through its cache to the home node. The home node sends updates to the other processors sharing the cache block, and a message to the writing processor containing the number of acknowledgements to expect. Sharing processors update their caches and send an acknowledgement to the writing processor. Since we assume release consistency, the writing processor does not have to wait for update acknowledgements before continuing execution; the processor only stalls waiting for acknowledgements at a lock release point. Under this protocol, blocks are evicted from the cache only due to replacements.

Our WU implementation includes two optimizations. First, when the home node receives an update for a block that is cached by the writing processor, the acknowledgement of the update instructs the processor to retain future updates since the data is effectively private. Second, when a parallel process is created by *fork*, we flush the cache of the parent's processor, which eliminates useless updates to data initialized by the parent but not needed subsequently by that processor.

The simulator implements a full-map directory for controlling the state of each block of memory. Each node contains the directory for the memory associated with that node.

3.2 Workload

Our application workload consists of four parallel programs: `Mp3d`, `Barnes`, `BLU`, and `SOR`. `Mp3d` is a wind-tunnel airflow simulation of 30000 particles for 5 steps. `Barnes` is an N-body application that

Application	Shared Refs	Shared Reads (% of shared refs)	Shared Writes (% of shared refs)
Mp3d	5.1 M	60.1 %	39.9 %
Barnes	19.0 M	97.5 %	2.5 %
BLU	47.2 M	89.6 %	10.4 %
SOR	20.7 M	84.5 %	15.5 %

Table 1: Memory reference characteristics.

simulates the evolution of 2K bodies under the influence of gravitational forces for 4 time steps. Mp3d and Barnes are part of the SPLASH suite [13]. BLU is an implementation of the blocked right-looking LU decomposition algorithm presented in [2] on a 384 × 384 matrix. SOR performs the successive over-relaxation of the temperature of a metal sheet represented by two 384 × 384 matrices. Table 1 summarizes the distribution of shared references in our applications on a 32-processor machine.

As is the case with similar studies, simulation constraints prevent experimentation with "real life" input data sets. Simply reducing the input size to manageable levels without changing the cache size could produce unrealistic results however. Therefore the input data sizes used for our applications were chosen in tandem with our choice of cache size. We first determined input sizes that could be simulated in a reasonable amount of time, and then experimented with various cache sizes for those data sets. The cache size we ultimately selected, 64 KB, was chosen so as to avoid too heavy an emphasis on replacement misses; this cache size is the smallest that holds the working set of processors for our applications.

4 Experimental Results

This section presents the results of our categorization for our application suite under the coherence protocols we just described and different cache block sizes.

4.1 Cache Miss Categorization under the WI Protocol

The vast majority of the traffic of applications under a WI protocol can be completely categorized by our classification of data-transferring cache misses (eviction, cold start, and true and false sharing). Exclusive request "misses" are presented for completeness.

Figures 6-9 present the miss rates for each of our applications under the WI protocol, as a function of block size. The number at the top of each column represents the percent of all references to shared data that result in a miss; within a column misses are classified as either eviction, cold start, exclusive request, true sharing, or false sharing misses.

Figure 6 shows the miss behavior of Barnes. Even though the working set of a processor fits in its cache, the eviction miss rate is still a problem due to limited spatial locality and to the mapping of addresses in direct-mapped caches. The minimum miss rate occurs with 64-byte blocks; larger blocks increase the number of eviction and false sharing misses. The other categories of misses decrease with an increase in block size.

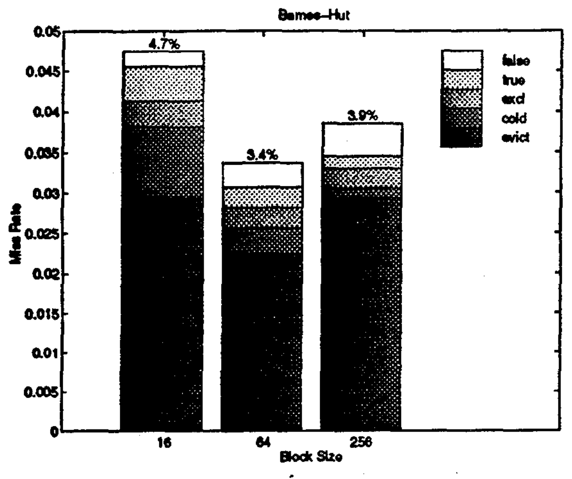

Figure 6: Miss rate of Barnes under WI

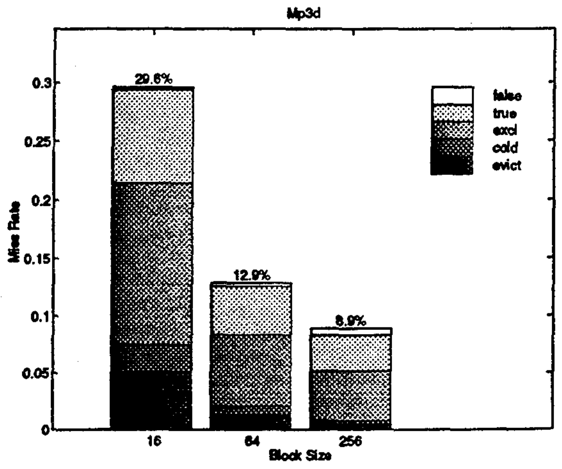

Figure 7: Miss rate of Mp3d under WI

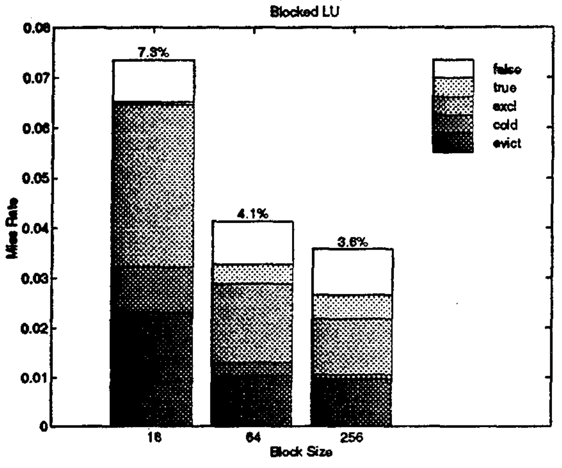

Figure 8: Miss rate of BLU under WI

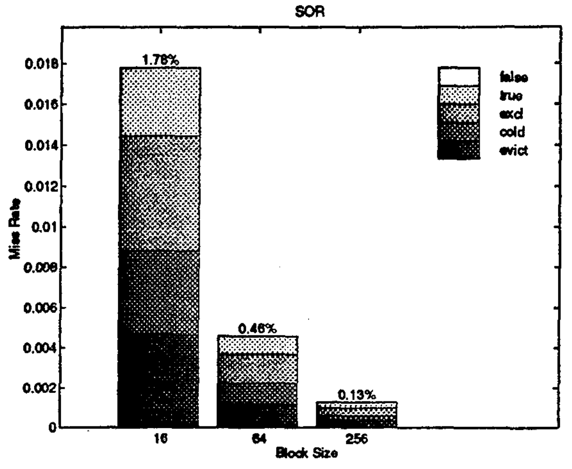

Figure 9: Miss rate of SOR under WI

As seen in figure 7, increasing the block size between 16 and 256 bytes results in a decrease in the miss rate of Mp3d. The composition of the miss rate differs markedly from Barnes. For Mp3d, exclusive requests are the most important source of misses, while replacements represent a small percentage of the total miss rate. Note also that the miss rate of Mp3d is high regardless of block size, and in all cases is dominated by sharing-related misses.

Figure 8 presents the miss rate behavior of BLU. As in Mp3d, the sharing-related misses dominate the miss rate of this program. For the first time we can see significant amounts of false sharing, which is introduced with 16-byte cache blocks and remains fairly constant with larger cache blocks. Despite the false sharing, the minimum miss rate is achieved with large cache blocks (256 bytes).

Figure 9 shows the miss rate of our SOR application as a function of the block size. As seen in the figure, SOR exhibits the ideal miss rate behavior; quadrupling the block size continualy cuts the miss rate in forths. Exclusive requests and replacement misses are the most important contributors to the miss rate of this program independently of the block size.

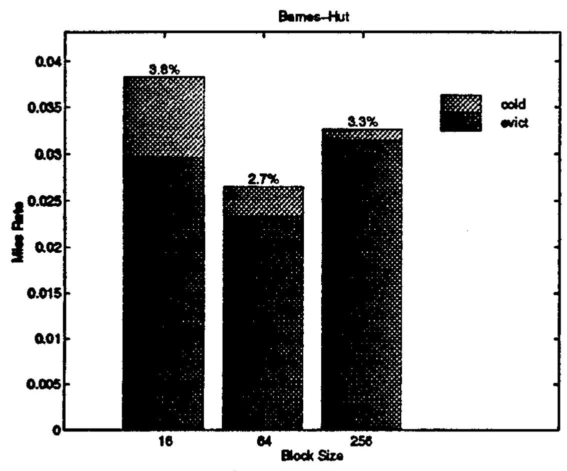

Figure 10: Miss rate of Barnes under WU

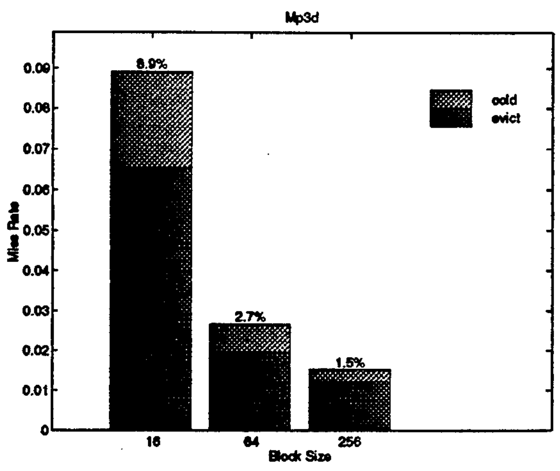

Figure 11: Miss rate of Mp3d under WU

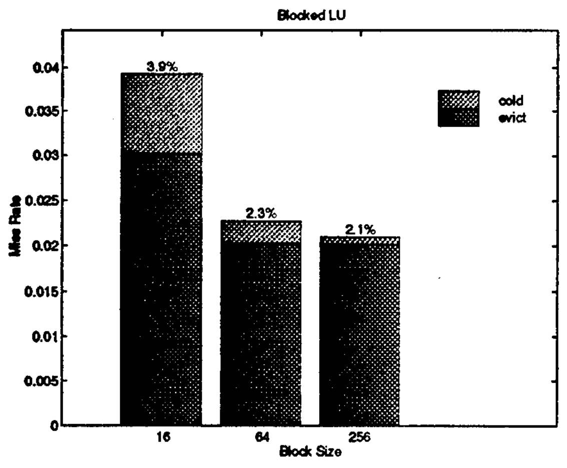

Figure 12: Miss rate of BLU under WU

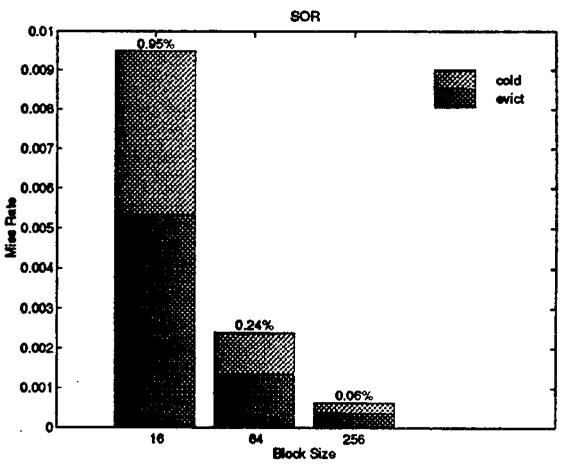

Figure 13: Miss rate of SOR under WU

4.2 Cache Miss Categorization under the WU Protocol

Figures 10-13 present the miss rates for each of our applications under the WU protocol, as a function of block size. Again the number at the top of each column represents the percent of all references to shared data that result in a miss. Since a pure WU protocol eliminates all sharing-related misses, each column in the graphs is divided in eviction and cold start misses.

Figure 6 shows the miss behavior of Barnes under our WU protocol. As seen in the figure, the overall behavior of the miss rate as a function of the block size remains the same as under the WI protocol, due to the overwhelming dominance of the U-shaped replacement miss rate. Due to the elimination of the sharing-related misses, Barnes exhibits somewhat lower miss rates under WU than under WI. Note that the WU miss rates are almost exactly the same as the sum of the eviction and cold start miss rates observed in the WI case.

As seen in figure 11, Mp3d does not follow this same trend. Although the elimination of sharing-

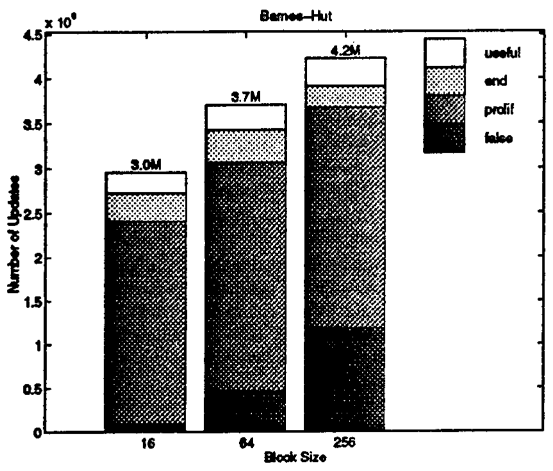

Figure 14: Update transactions of Barnes

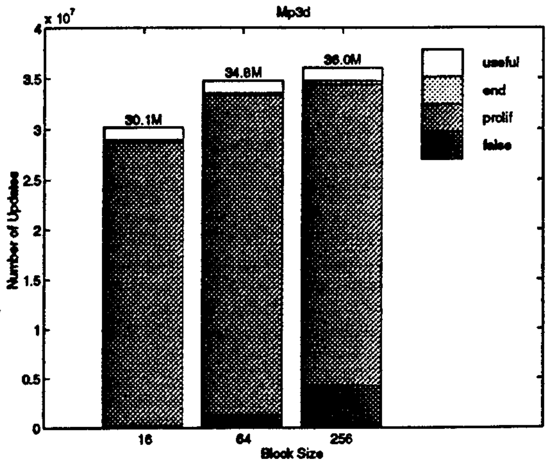

Figure 15: Update transactions of Mp3d

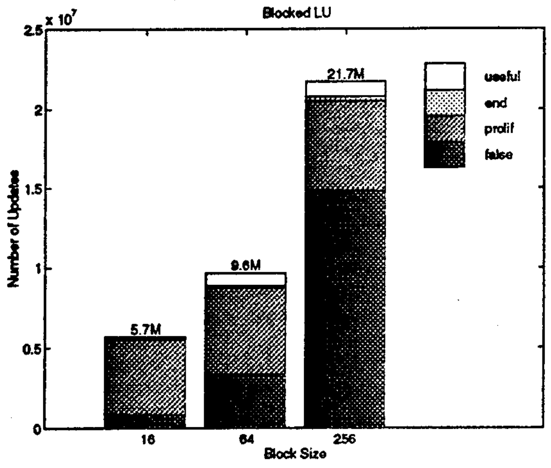

Figure 16: Update transactions of BLU

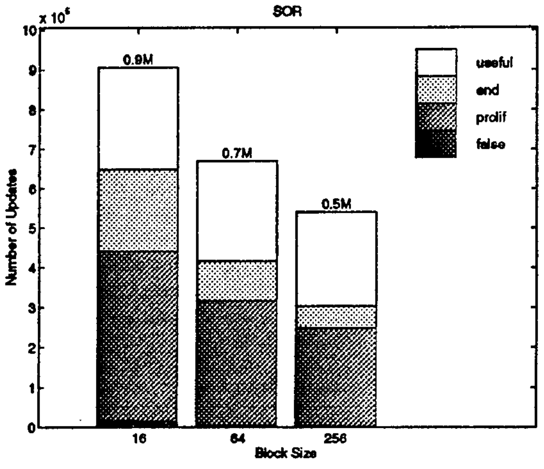

Figure 17: Update transactions of SOR

related misses does decrease the miss rate significantly, the overall miss rate under WU is much larger than the sum of the eviction and cold start miss rates under the WI protocol. The reason for this effect is that update-based protocols usually keep a larger amount of data in a processor's cache, as writes by other processors sharing the same data do not cause invalidations that could free space in the cache. The overall behavior of the miss rate as a function of the block size is the same as under WI. However, the miss rate improvements achieved with increases in the size of blocks is much larger under WU than under WI.

Figure 12 shows that, similarly to Mp3d, BLU exhibits much lower miss rates and an inflated replacement miss rate under WU in comparison to the WI protocol. Again, the overall behavior of the miss rate as a function of the block size is roughly the same as under WI.

As seen in figure 13, SOR also exhibits significant miss rate improvements through the use of a WU coherence protocol, due to the elimination of exclusive request and true sharing misses which contributed heavily to the WI miss rate of this application. Similarly to the miss rates of SOR under the WI protocol, we see that increasing the block size results in a proportional decrease in miss rate.

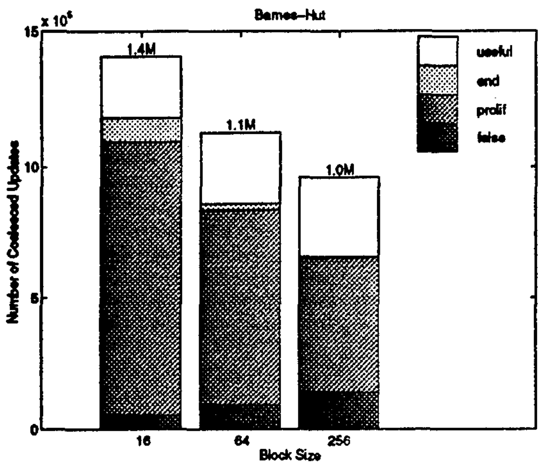

Figure 18: Coalesced update transactions of Barnes

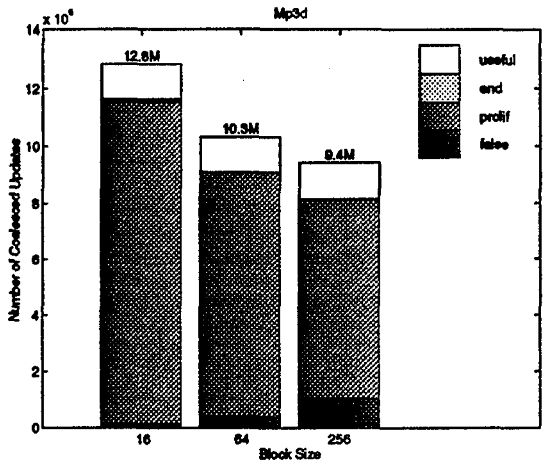

Figure 19: Coalesced update transactions of Mp3d

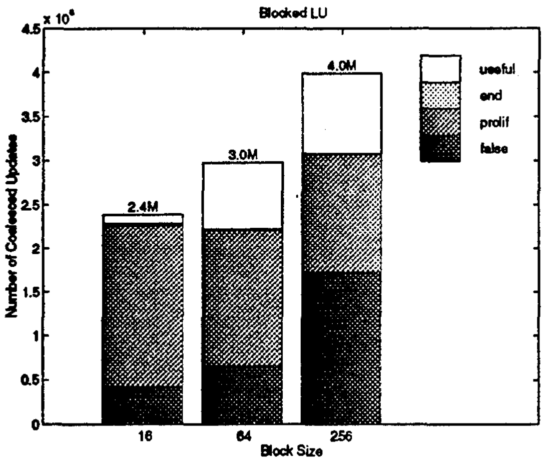

Figure 20: Coalesced update transactions of BLU

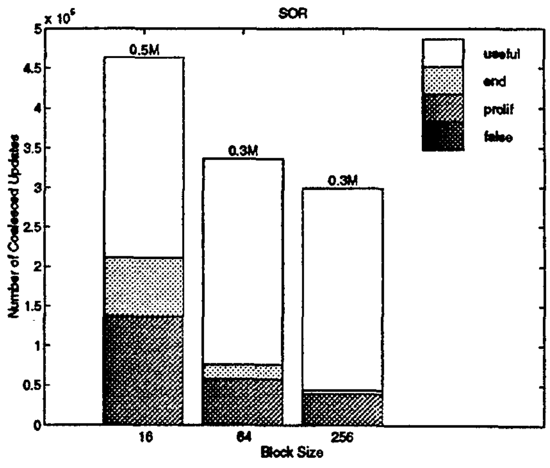

Figure 21: Coalesced update transactions of SOR

4.3 Update Transaction Categorization

Our categorization of update transactions (and their associated acknowledgements) can be combined to the classification of cache misses presented in the previous section to represent the vast majority of the communication traffic found under WU protocols.

Figures 14-17 present our categorization of update transactions for each of our applications, as a function of block size. The number at the top of each column represents the total number of update transactions; within a column updates are classified as either false sharing, proliferation, end, or true sharing (useful) updates.

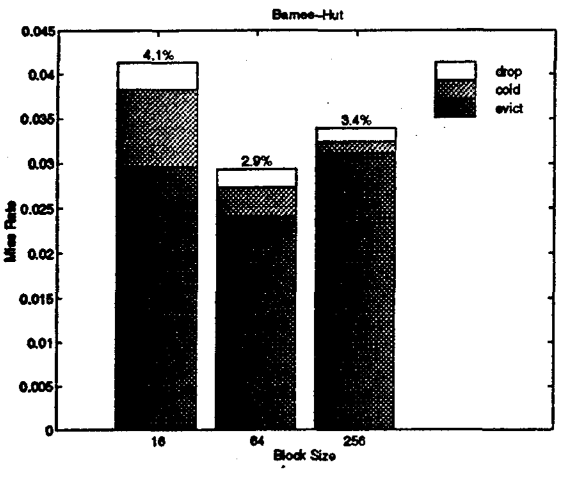

Figure 22: Miss rate of Barnes under Hybrid Protocol

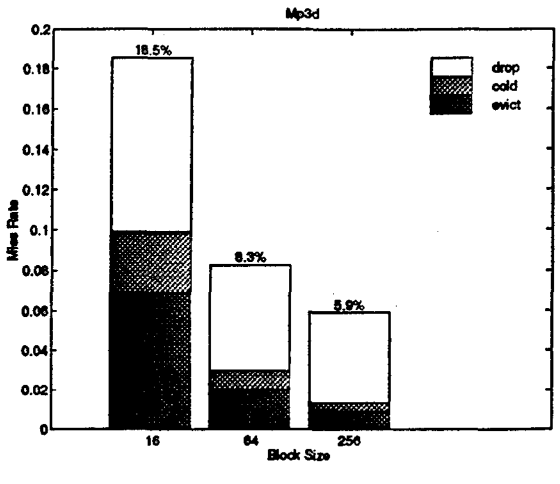

Figure 23: Miss rate of Mp3d under Hybrid Protocol

We can see from these figures that the number of useless updates is extremely high: 90% or more of all updates sent during execution of Barnes, BLU, and Mp3d are useless, while between 56% and 72% of all updates in SOR are useless, depending on the block size. Despite the two optimizations described in section 3, which eliminate useless updates for data that is effectively private and for data that is initialized by a parent process prior to a *fork*, the vast majority of the remaining updates are still useless.

Proliferation updates dominate in three of the programs (Barnes, Mp3d, and SOR) independently of the block size, while false updates are more numerous in the fourth (BLU) with the largest cache block we studied. In most cases, increasing the block size causes a substantial increase either in the number of proliferation updates or in the number of false sharing updates, due to the fact that, with larger blocks, sharing becomes more widespread and processors rarely drop copies of data they no longer need. The exception here is SOR; it exhibits almost perfect spatial and processor locality and, as a result, an increase in the block size does not increase the degree of sharing of the program.

4.4 Coalesced Update Transactions Categorization

This section presents the results of our categorization of update transactions in the presence of coalescing write buffers, as a function of the cache block size. Figures 18-21 present our results for each of our applications. The number at the top of each column represents the total number of coalesced update transactions; within a column coalesced updates are classified as either false sharing, proliferation, end, or true sharing (useful) updates.

Comparing these figures with the ones in the previous section, we see that, while coalescing increased the percentage of useful update transactions for all applications independently of the block size, it also changed the overall update behavior of two applications completely (Barnes and Mp3d). For these two applications, increasing the block size results in a reduction of the total number of update messages. This is due primarily to the significant decrease in the number of proliferation updates.

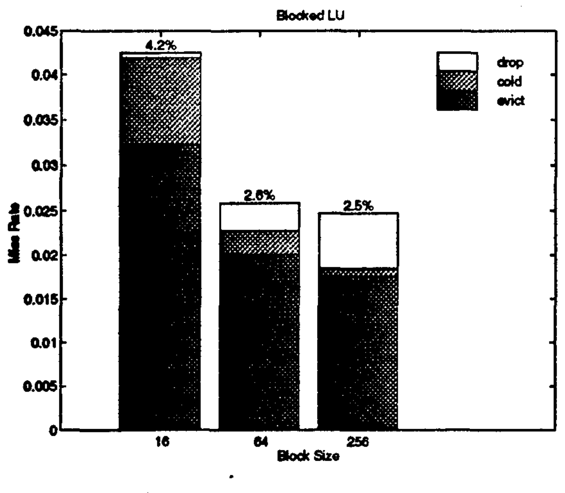

Figure 24: Miss rate of BLU under Hybrid Protocol

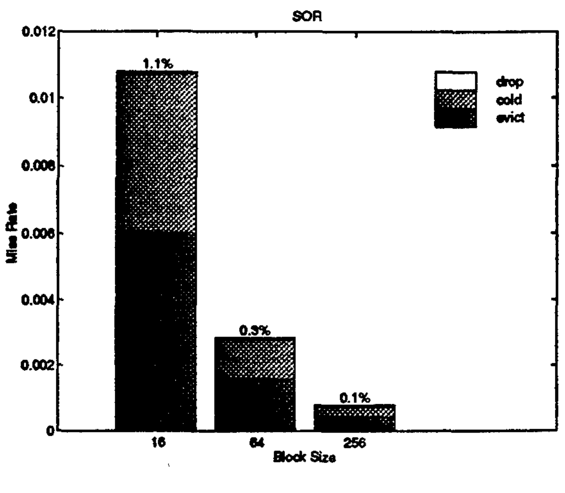

Figure 25: Miss rate of SOR under Hybrid Protocol

4.5 Cache Miss Categorization under the Hybrid Protocol

Our implementation of the hybrid protocol described in section 2.4 associates a counter with each cache block and invalidates the block when the counter reaches a threshold of 4 updates without intervening accesses by the local processor. When the threshold is reached, the node sends a message to the block's home node asking it not to send any more updates to the node. References to a cache block reset the counter to zero. The results we present here assume coalescing write buffers.

Figures 22-25 present the miss rates for each of our applications under the our hybrid protocol, as a function of block size. Again, the number at the top of each column represents the percent of all references to shared data that result in a miss. Since cache misses can result from incorrectly dropping blocks, each column in the graphs is divided in eviction, cold start, and drop misses.

The results shown in the figures demonstrate that invalidating cache blocks may or may not significantly increase the miss rates found under the WU protocol. Two applications exhibit very slight increases in miss rates due to bad block drops (Barnes and SOR), while the other applications suffer more significantly, especially Mp3d. However, even these applications exhibit lower miss rates under the hybrid protocol than under a WI protocol.

4.6 Update Transaction Categorization Under the Hybrid Protocol

Figures 26-29 present the number of coalesced update transactions for each of our applications under the our hybrid coherence protocol, as a function of block size. Again, the number at the top of each column represents the total number of update transactions; within a column updates are classified as either false sharing, proliferation, end, or true sharing (useful) updates.

The results shown in the figures indicate that usually the percentage of useful update messages increases significantly under the hybrid coherence protocol. In addition, the applications exhibit significant reductions in the total number of update transactions, especially for the larger block sizes. Mp3d exhibits the greatest reductions in the number of update transactions: around 78% for all block sizes.

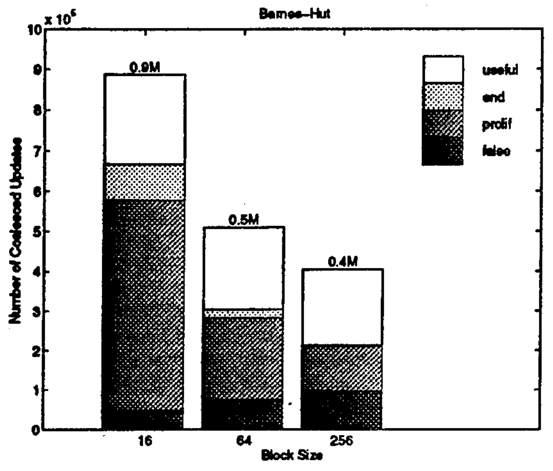

Figure 26: Coalesced update transactions of Barnes under Hybrid Protocol

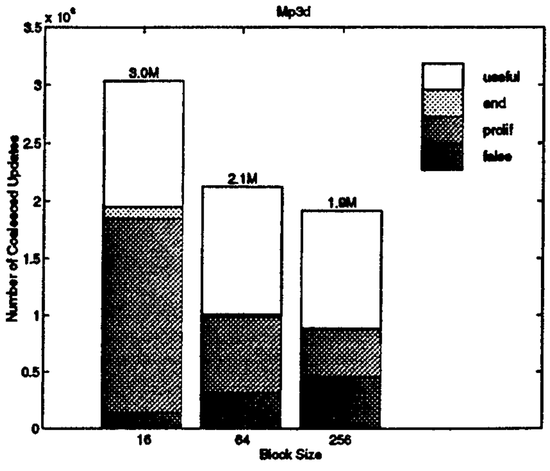

Figure 27: Coalesced update transactions of Mp3d under Hybrid Protocol

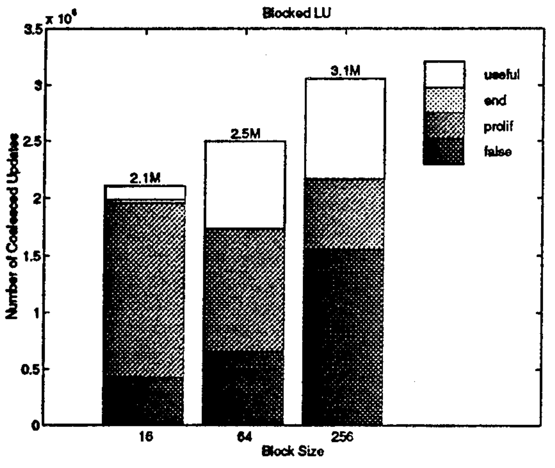

Figure 28: Coalesced update transactions of BLU under Hybrid Protocol

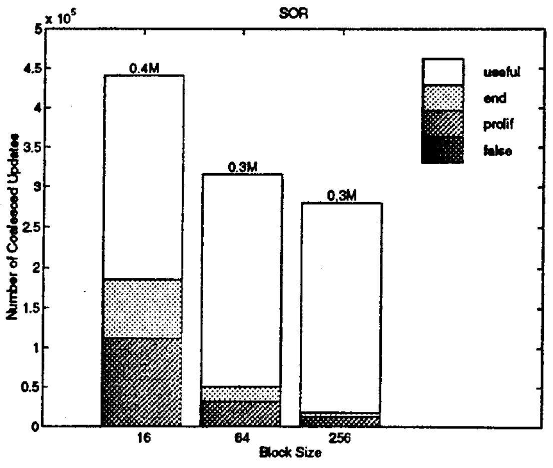

Figure 29: Coalesced update transactions of SOR under Hybrid Protocol

5 Related Work

Most of the work dealing with multiprocessor communication does not seek to accurately characterize the source of the observed traffic. Notable exceptions, such as [7], have focused on measuring the amount of each type of communication (requests, data, and coherence), and coupling these measures with information about the reference and sharing patterns of application programs. Those studies have invariably been concentrated on WI protocols.

Although, very few papers have studied the usefulness of the communication traffic observed when running parallel programs, there has been some research on developing algorithms that attempt to characterize false sharing under invalidation-based protocols. Torrellas *et al* [15] present a scheme that classifies misses as true sharing if they access a word that has been modified before and they are also misses on a single word cache line system. All other misses are classified as false sharing. Eggers *et al* [5] present a simpler scheme that classifies a miss as true sharing if it is a miss on a word that has been modified since the last invalidation. None of the above approaches take into account the prefetching effect of long cache lines and as a consequence may overestimate the number of false sharing misses. The scheme proposed by Torrellas partially compensates for this overestimation by requiring the additional constraint that a true sharing miss be also a miss in a single word cache line system.

Dubois *et al* [4] propose a scheme that delays classifying a miss until the subsequent invalidation of the block missed on, or the end of the program (whichever happens first). They show that their scheme accurately captures the intuition behind false sharing and remedies the problems encountered in the first two approaches. Our algorithm for classifying cache misses is a straightforward extension of the one presented in [4]. Bolosky and Scott [1] also present a scheme that estimates false sharing and the overhead of reducing it by reducing the cache line size (a cache line size of one has no false sharing). None of the above approaches attempt to incorporate an eviction miss detection scheme into the classification algorithm.

Write-update classification schemes have received almost no attention. Khera *et al* [9] present an architecture independent analysis of false sharing that attempts to characterize false sharing independent of architecture. However the approach is statistical in nature and encompasses several assumptions; such estimates can be extremely inaccurate in more practical scenarios.

6 Conclusions

In this paper we have presented algorithms for categorizing the major sources of communication under both WI and WU protocols. The vast majority of the communication traffic found under an invalidate-based protocol is due to data-related transactions caused by cache misses. For WI protocols, we have extended a well-known algorithm for classifying cache misses that deals with finite sized caches.

Under an update-based protocol, the vast majority of the communication traffic is caused by cache misses and update transactions (and their associated acknowledgements). We introduced an algorithm that accurately characterizes update transactions under a WU protocol (extensions of this algorithm considered coalescing write buffers and competitive update strategies).

Classification schemes, such as the ones proposed in this paper, provide insight into the reference and sharing patterns and into the amount of useless traffic generated by parallel applications. In particular, the experiments described in this paper allowed us to observe the effect of changes in block size on the composition of the miss rate of applications under the different coherence protocols we considered. Our results also revealed the amount of useless traffic entailed by each coherence protocol, exposing the high percentage of unnecessary update transactions under a WU protocol. Finally, our experiments illustrated the beneficial impact of two architectural variations of update-based protocols on the amount and type of coherence traffic generated by parallel programs.

We believe that our algorithms provide the necessary framework for classification schemes for any coherence protocol. Two reasons support this claim: a) as we have shown in this paper, our algorithms can be easily extended to account for possible coherence protocol and architectural variations; and b) although the algorithms deal with numerous hardware features, our categorization is widely applicable and can be easily simplified for use in less detailed environments.

Acknowledgements

We would like to thank Wei Li and our colleagues who participated in his seminar (Michal Cierniak, Mike Marchetti, Maged Michael, and Mohammed Zaki) for their help in developing the ideas that led to the algorithms presented in this paper. We would also like to thank Tom LeBlanc and Michael Scott for their advice and comments on earlier drafts of this work.

References

[1] W. J. Bolosky and M. L. Scott. False Sharing and its Effect on Shared Memory Performance. In *Proceedings of the Fourth USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 57–71, September 1993. Also available as MSR-TR-93-1, Microsoft Research Laboratory, September 1993.

[2] K. Dackland, E. Elmroth, B. Kagstrom, and C. V. Loan. Parallel Block Matrix Factorizations on the Shared-Memory Multiprocessor IBM 3090 VF/600J. *The International Journal of Supercomputer Applications*, 6(1):69–97, Spring 1992.

[3] *Alpha Architecture Handbook.* Digital Equipment Corporation, 1992.

[4] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, pages 88–97, San Diego, CA, May 1993.

[5] S. J. Eggers and T. E. Jeremiassen. Eliminating False Sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages I:377–381, St. Charles, IL, August 1991.

[6] J. R. Goodman. Using Cache Memory to Reduce Processor/Memory Traffic. In *Proceedings of the Tenth International Symposium on Computer Architecture*, pages 124–131, June 1983.

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

[8] N. Jouppi. Cache Write Policies and Performance. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[9] V. Khera, R. P. LaRowe Jr., and C. S. Ellis. An Architecture-Independent Analysis of False Sharing. CS-1993-13, Department of Computer Science, Duke University, October 1993. Also TR 93-006, Center for High Performance Computing.

[10] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 148–159, Seattle, WA, May 1990.

[11] E. M. McCreight. The Dragon Computer System: An Early Overview. In *Proceedings of the Nato Advanced Study Institute on Microarchitecture of VLSI Computers*, July 1984.

[12] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the Eleventh International Symposium on Computer Architecture*, pages 348–354, May 1984.

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

[14] C. P. Thacker and L. C. Stewart. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988. Originally presented at the *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987.

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

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