Appears in the 13th Workshop on Interaction between Compilers and Computer Architecture (Interact-13)
held in conjunction with 15th IEEE International Symposium on High-Performance Computer Architecture, February 2009.

Coherence Miss Classification for Performance Debugging in Multi-Core Processors

Guru Venkataramani Georgia Tech
Christopher J. Hughes Intel Corp
Sanjeev Kumar Facebook Inc
Milos Prvulovic Georgia Tech

guru@cc.gatech.edu

Abstract

Multi-core processors offer large performance potential for parallel applications, but writing these applications is notoriously difficult. Tuning a parallel application to achieve scalability, referred to as performance debugging, is often more challenging for programmers than conventional debugging for correctness. Parallel programs have several performance related issues that are not seen in sequential programs. In particular, increased cache misses triggered by data sharing (coherence misses) are a challenge for programmers. Data sharing misses can stem from true or false sharing and the solutions for the two types of misses are quite different. Therefore, to minimize coherence misses, it is not just sufficient for programmers to only identify the source of the extra misses. They should also have information about the type of coherence misses that are hurting performance.

In this paper, we propose a programmer-centric definition of false sharing misses for use in performance debugging. We describe our algorithm to classify coherence misses based on this definition, and explore a practical and low cost solution that keeps no state at all. We find that the low cost solution can suffer from considerable inaccuracy that might mislead programmers in their performance debugging efforts.

1. Introduction

As architects design multi-core and many-core architectures, programmers take advantage of the available resources and develop applications with better performance. The end-users are enticed by the newer applications and invest in buying new hardware. However, if the programmers cannot effectively utilize multiple cores on the chip, they fail to deliver newer and scalable applications to the end-user. As a result, it is important for architects to focus on techniques that will aid programmers in their performance debugging efforts.

To a certain degree, hardware manufacturers already understand that making processors with increased raw performance is not enough. They want customers to see a real performance difference when running their applications. This is not a trivial task for application programmers and therefore, many hardware manufacturers currently provide support through hardware performance counters [1]. Performance counters help programmers to find certain common performance bottlenecks such as branch mispredictions, cache misses, etc. It is significantly harder to realize the performance potential of multi-core and many-core processors and additional support will be needed for performance debugging in such processors. Ideally, this support would include performance counters for additional multi-core and many-core events, and hardware to detect those events and provide feedback to programmers, dynamic optimizers or compilers.

Hardware support for performance debugging brings significant value given the wide variety of software-only performance debugging tools available. Tools that interact with the performance counters are considered very useful because they can collect information on a variety of performance limiters and present it in a usable way for programmers. They are also very fast and accurate. In contrast, software-only tools typically measure a small set of events since they must have reasonable trade off between execution time overhead and the amount of information to be collected. Such tools are generally slow even when only measuring a small number of events (e.g., MemSpy [12] incurs up to 57.9x slowdown over native execution and SM-prof [4] reports up to 3000x slowdown depending on the amount of shared data references). Above all, software tools are not reliably accurate because in many cases they tend to perturb the program execution with the program's natural execution (e.g., via software instrumentation). Building a performance debugging tool that is fast, accurate, and that can still measure a number of events of interest is extremely valuable. Therefore, we explore additional hardware support that can significantly aid multi-core and many-core performance debugging.

1.1. Coherence Misses

Scalability of an application can be limited by the amount of parallelism available in the underlying algorithm or by the hardware that executes the application. Some of the important hardware-related scaling limiters are those that increase cache miss rates when using multiple threads. To eliminate or alleviate these cache misses, the application developer must identify the underlying cause of these misses to make necessary modifications to the code. In order to do so, information

on the type of miss is necessary because different fixes are applied to the code for different types of cache misses. Misses caused by true and false sharing, collectively known as coherence misses, are especially problematic (See Section 2). The two types of coherence misses often have similar symptoms, but the methods for alleviating them are very different. False sharing is often easily addressed by separating affected data in memory, e.g. by adding padding. True sharing misses are typically reduced only through more extensive changes, such as changes to assignment of tasks to threads or using a different parallel algorithm. Both types differ from other types of cache misses (e.g., capacity and conflict) that are typically addressed by trying to fit data into the cache more effectively by managing the working set.

Coherence misses do not occur in uniprocessors, so, many programmers are not familiar with them. To understand the cause of these misses the developer must have a working knowledge and understanding of the coherence protocol and how it interacts with caches. Without suitable tools, programmers need to guess whether an increase in misses is caused by coherence and whether such misses are caused by true or false sharing. In our experience, false sharing often surprises programmers—without sufficient knowledge of the underlying hardware, the presence of these misses is mysterious and even frustrating. Instead, profiling infrastructure should detect true and false sharing misses, attribute it to particular points in the code, and report them to the programmer. With such an infrastructure, the developer can focus on solving the performance problem rather than figuring out the hardware mechanisms that cause it.

1.2. Our Contributions

This paper makes the following contributions:

1. We provide several real-world examples of false sharing and illustrate the performance impact on real applications.

2. We present a programmer-centric definition of false sharing misses based on definition provided by Dubois et al. [7] and develop an oracle algorithm (o-FSD) to detect false sharing based on this definition.

3. We provide two different hardware implementations for false sharing detection with very different cost and accuracy tradeoffs. We evaluate these implementations against our o-FSD algorithm using Splash-2 [18] and PARSEC [3] benchmarks.

2. False Sharing and its implications

In this section, we explore a programmer-centric definition for false sharing and true sharing misses and then show several real-world examples where false sharing misses occur and how removing these misses improves scalability.

2.1. Definition of False Sharing

In cache coherent CMPs, data sharing between threads primarily manifests itself as coherence misses which can further be classified as true sharing and false sharing misses.

True sharing misses are a consequence of actual data sharing amongst cores and is intuitive to most programmers—e.g., a consumer (reader) of the data must suffer a cache miss to obtain the updated version of the data from the producer (writer). The scalability issues stemming from true sharing misses can typically be addressed only by changing the algorithm, distribution of the work among cores, or synchronization and timing.

In contrast, false sharing misses are an artifact of data placement and a cache block holding multiple data items. Scalability issues arising from false sharing are often relatively easy to alleviate by changing alignment or adding padding between affected data items. A false sharing miss occurs if a block contains two data items (A and B), core M accesses item A, another core N uses item B and invalidates the block from M's cache, and then core M accesses item A again. The resulting cache miss is a false sharing miss because that access would be a cache hit if A and B were in different cache blocks. In fact, the definition of false sharing can be made more precise. Specifically, if after the coherence miss on an item A, core M accesses item B before the block is evicted or invalidated again, the miss is in fact a true sharing miss. As shown in Figure 1, in this situation, the cache miss is not avoided by placing A and B in separate blocks, and hence, the miss on A is not a false sharing miss. This definition of false sharing is based on Dubois et al. [7], and it differs from earlier definitions [8, 17] in that it attempts to classify coherence misses according to whether they are "necessary" (true sharing) or "unnecessary" (false sharing). We adopt this definition as a baseline because it more accurately captures whether or not the miss can be eliminated (not just traded for another miss) by separating data items into different blocks which, in the end, is what really matters for the programmer's performance debugging efforts.

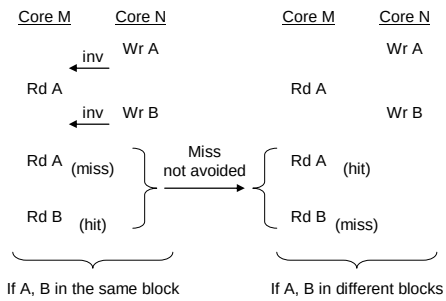

Figure 1. The miss on A is a not a false sharing miss — it is only replaced by another miss if A and B are placed in separate blocks.

```
// Each thread processes its own
// sequence of input elements
int partial_result[NUM_THREADS];
// This is the work done in parallel
...
int input_element=...;
partial_result[thread_id] += input_element;
...
```

Figure 2. A parallel reduction showing false sharing on an array of counters (one-per-thread). The merging of partial results is omitted for brevity.

```
// Count occurrences of values in parallel
// Each thread processes its own range of
// input elements, updating the shared
// occurrence count for each element's value

#define MAXIMUM 255
int counter_array[MAXIMUM+1];
// This is the work done in parallel
...
int input_element=...;
  counter_array[input_element]++;
...
```

Figure 3. A parallel histogram computation illustrating false sharing on an indirectly accessed array. Locking of counter_array elements is omitted for brevity.

2.2. Real-World examples of False Sharing limiting Scalability

We provide examples, taken from code written by experienced programmers, to illustrate some common situations where false sharing occurs and its sometimes devastating impact on parallel scalability.

Our first example involves an array of private counters or accumulators (one per-thread), which is often used when parallelizing reductions as shown in Figure 2. There is no true sharing in this code because each thread is reading and writing a unique array element. False sharing occurs when two threads' counters lie in the same cache block. This kind of code is common in web search, fluid simulation, and human body tracking applications. A common fix is to add padding around each counter. We use a real-world example of a loop in *One_Newton_Step_Toward_Steady_State_CG_Helper_II()* function in facesim from the PARSEC-1.0 benchmark suite. The benchmark authors spent multiple days identifying false sharing from this loop as the primary source of performance problems. We ran the facesim benchmark with the native input on an 8-core Intel Xeon machine and observed that without padding, false sharing limits the benchmark's parallel scaling to 4x. After adding padding, the benchmark achieves linear scaling (8x). A profiling tool that automatically identifies and reports false sharing misses would have greatly helped the programmer.

```
// The task of each thread is to update
// one row of the grid
...
for (i = (iteration\%2);i < width;i += 2) {
  float val = grid[task_id  ][i  ] +
              grid[task_id  ][i-1] +
              grid[task_id  ][i+1] +
              grid[task_id-1][i  ] +
              grid[task_id+1][i  ];
  grid[task_id][i] = val / 5.0;
}
...
```

Figure 4. A red-black Gauss-Seidel-style array update showing false sharing on a finely partitioned array.

Our second example involves an indirectly accessed data array, as shown in Figure 3 and often occurs in histogram computation, used in image processing applications and in some implementations of radix sort. The pattern of indirections is input-dependent, so the programmer and compiler cannot predetermine how many accesses will occur to each element, and which threads will perform them. This example involves both true and false sharing. True sharing occurs when two threads update the same element. False sharing occurs when two threads access two different elements in the same cache block, which occurs much more frequently. For example, with 64-byte blocks and 4-byte elements, a block contains 16 elements; With a completely random access pattern, false sharing would occur 15 times more likely than true sharing. A common fix is to either add padding around each element or use privatization (use a separate array for each thread and then merge partial results at the end). Without privatization, a histogram benchmark from a real-world image processing application achieves only a 2x parallel speedup when run on a 16-core Intel Xeon machine. With privatization, the benchmark achieves near-linear scalability.

Our final example of false sharing involves finely partitioned arrays, such as one might see in red-black Gauss-Seidel computation as shown in Figure 4. In many applications, a data array is partitioned such that each partition will only be written to by a single thread. However, when updating elements around the boundary of a partition, a thread sometimes needs to read data across the boundary (i.e., from another partition). In our example, synchronization on each element is avoided by treating the data as a checkerboard, with alternating red and black elements. Even-numbered passes update red cells, and odd-numbered passes update black cells. An update involves reading the Manhattan-adjacent neighbors, which are guaranteed to be a different color than the cell being updated. Thus, even if those neighbors belong to another partition, they will not be updated during the current pass.

Both true and false sharing occurs in this example. The first time a thread accesses a cache block across a partition

boundary, it is reading data written by another thread during the previous pass. Thus, it incurs a true sharing miss. However, if that other thread is actively updating elements in the same cache block, this will trigger additional misses that are all due to false sharing. This situation is most likely when partitions are small; for example, if a parallel task is to update one row, and the tasks are distributed to threads round-robin. This kind of computation is common in scientific codes (i.e., applications that involve solving systems of differential equations). We ran an early real-world implementation of red-black Gauss-Seidel with the above task distribution on an 8-core Intel Xeon processor. Parallel scaling is limited to 3x. Padding around each cell can eliminate false sharing, but with significant loss in spacial locality. A better solution is to group multiple rows together to be processed by the same thread, or to use a two separate arrays, one for red and one for black cells. In our case, using separate arrays and grouping rows improved the scaling to almost 6x.

3. Coherence Miss Classification

In this section, we first describe the relatively simple mechanism for distinguishing coherence misses from non-coherence misses, then discuss an oracle False Sharing Detector (o-FSD) mechanism that further classifies coherence misses into those caused by false sharing and those caused by true sharing. Finally, we show why o-FSD is impractical to implement in real hardware.

3.1. Identification of Coherence Misses

Coherence misses can be distinguished from other (cold, coherence, or conflict) misses by checking if the cache block is already present in the cache. For non-coherence misses, the block either never was in the cache (cold miss) or was replaced by another block (capacity or conflict miss). In contrast, a coherence miss occurs when the block was *invalidated* or *downgraded* to allow another core to cache and access that block. Coherence misses are easily detected with a minor modification to the existing cache lookup procedure. A cache miss is detected as a coherence miss if a block does have a matching tag but does not have a valid state. Conversely, if no block with a matching tag is found, we have a non-coherence miss.

This relatively simple mechanism to detect coherence misses can err in two ways. First, on power-up or after a page fault, the state of a block's state is set to invalid, but the tag need not be initialized. This could sometimes result in a tag accidentally matching to that of a requested block. However, this would be quite rare and random enough that they do not attribute in significant numbers to any particular piece of code. Second, cache replacement policy could prioritize replacement of invalidated blocks, which can destroy the evidence of a coherence miss. For highly contended blocks involved in sharing patterns, there is little time for the block to be replaced between the invalidation and the subsequent miss, so replacement priority is unlikely to obscure enough coherence misses to hide a scalability problem. It should also be noted that the inaccuracy caused by replacement priority is very costly to avoid: it requires the cache to track the tags of blocks that *would be* in the cache were it not for replacement priority, e.g. using a duplicate set of tags.

3.2. Oracle False Sharing Detector (o-FSD)

Using the definitions of false and true sharing from Section 2, coherence misses can be classified into false sharing misses and true sharing misses as described in our algorithm shown in Figure 5. Coherence miss classification involves two parts: 1) From the time a cache block is invalidated or downgraded in a core's cache until the time when coherence miss happens, we need information about which words were read from or written to by other cores. 2) From the time of coherence miss until when the block is invalidated/downgraded/replaced again, we need to determine whether an access to any word in the block overlaps with an access by another core. True sharing is detected when a memory word is being produced and consumed by two different cores. If we do not detect any true sharing, then the coherence miss is a false sharing miss.

A read access is a true sharing access if the word's last write was not done by this core and if this core has not already read the word since it was last written by another core. A write access is a true sharing access if the word's last write was not done by this core or if it was read by another core since it was last written. A coherence miss is classified as a true sharing miss if any access by a thread, starting with the one that causes the miss and ending with the subsequent replacement or invalidation of the block, is a true-sharing access. Conversely, the miss is categorized as a false sharing miss if no true sharing access occurs in this interval.

For implementing o-FSD, there are two major data structures: a) `global state` maintained for every word that tracks last writer core and subsequent readers until the next write. This tracks the reads and writes performed by all the cores in a central structure. Note that, tracking granularity is a matter of cost-performance tradeoff. In our experiments, we didn't observe significant coherence activity on sub-word accesses. Hence we chose word-granularity for our algorithm. We note that byte-granularity can be achieved with the same algorithm with relatively straightforward modifications. b) `local state` that is maintained for every private cache block stores the following information: 1) Two bit vectors which record reads and writes performed by other cores to the cache block prior to the coherence miss being classified. 2) Program Counter at the time of coherence miss for attribution to the program code that caused the miss. 3) Two flags, one to track whether the cache block suffered coherence miss and the other to track whether the all the accesses to the cache block are false sharing accesses. 4) Two bit vectors that track read and write accesses by local core to update

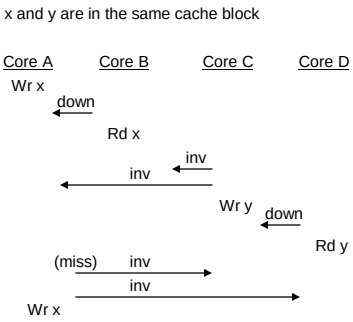

Figure 6. Example of multiple producers and multiple consumers for a cache block.

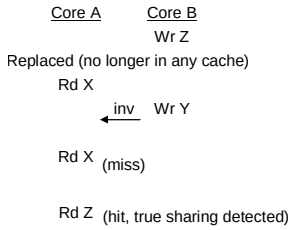

Figure 7. Accurate classification requires us to keep access information even for blocks that are no longer in any cache.

the global state at the time of cache block invalidation/downgrade/replacement. This is needed for other cores to perform coherence miss classification correctly.

Dubois et al. [7] have proposed an algorithm for detecting false sharing misses. In their scheme, the core producing the value notifies all the other cores (potential consumers) that a new value has been written to a word. When a core consumes that new value, it sets a local bit that denotes that the value has been read. When the producer suffers a coherence miss, it checks if any other core consumed the last generated value by checking the local caches of other cores that currently have a valid copy of the cache block.

Our o-FSD algorithm has two key differences with the algorithm proposed by Dubois et al. [7]. First, o-FSD classifies coherence misses both on the producer and consumer side whereas Dubois' scheme does coherence miss classification only on the producer side. In a *single producer-multiple consumer* pattern (See Gauss-Siedel example in Section 2.2), coherence misses are highly likely to happen in multiple cores (producer and the respective consumers). Dubois' scheme could underestimate the effect of false sharing in such situations and the programmer might consequently ignore or oversee the lines of code involved in such patterns because of lack of understanding of its impact. Second, Dubois' scheme checks only for read-sharers with valid blocks in their caches to gather information for determining the type of coherence miss on the writer side. This is problematic especially when there are multiple producers and multiple consumers sharing a cache block (See histogram example Section 2.2). Figure 6 shows an example where four cores A, B, C and D share a cache block. Core B consumes the value of x produced by core A and later on, core D consumes the value of y produced by core C. When core A writes to x again, it suffers coherence miss. Since Dubois' scheme would only check the information in core D which currently holds a 'valid' copy of the block, it will detect that x had not been consumed by core D and would declare false sharing. However, our o-FSD algorithm would have information about 'read x by core B' in its global state and hence, will correctly detect true sharing.

Additionally, we maintain the local state that records reads and writes by local core and update the global state only on replacements and invalidations. Dubois' scheme directly updates its global state on all accesses which is suitable for simulators but cannot be incorporated in real hardware.

3.3. Suitability of o-FSD for On-Line Miss Classification

There are a number of problems that make the oracle False Sharing Detector unsuitable for on-line implementation needed to drive hardware performance counters or other hardware performance debugging and attribution mechanisms. The most significant of these problems are:

1. The o-FSD has a high implementation cost which does not scale well. The *per-word* `global state` for the o-FSD structure shown in Figure 5 is $P + log_2 P$ where P is the number of cores. With 4 cores and 32-bit words this state represents a 19% storage overhead, and with 32 cores the storage overhead is already 116%.

2. The state needed for the o-FSD can be very large (requires keeping state for all words ever touched). Although, it may be tempting to not keep track state for a word that is no longer present in any core's cache. However, information about currently evicted blocks from cache might be relevant for classifying future coherence misses to these blocks. For example, to accurately classify the true-sharing miss to data item X on Core A in Figure 7, we must know that the subsequently accessed data item Z has last been written by Core B, even though the write on Z occurred long ago and the block was evicted from cache at some point between that write on Core B and then later read on Core A.

3. Changes are needed to the underlying cache coherence protocol and extra network traffic is required to update the o-FSD state and/or propagate it to the cores that need to classify their coherence misses. In particular, global state information for a memory word should be kept in a central repository. As seen in Figure 5, the coherence protocol needs to be modified to trigger reads and writes of the global state information at appropriate times. To enforce that, the central repository needs to ensure updates sent by the cores replacing/invalidating/-downgrading the cache block are reflected in the global state before the core suffering the coherence miss reads

```
//P=total number of cores, W=number of words in a cache block
//A=current data address, B=cache block holding address A
//c=current core

global state for every word in memory:
  Global_Writer_ID          //Last Writer ID (length=log(P) bits)
  Global_Reader_vector      //Readers since last write (length=P bits)
local state for each private cache block:
  Write_vector_of_others //Words modified by other cores (length=W bits)
  Read_vector_of_others  //Words read by other cores (length=W bits)
  Local_Write_vector     //Words modified by this core (length=W bits)
  Local_Read_vector      //Words read by this core (length=W bits)
  Miss_PC                //Program Counter causing coherence miss
  Coherence_Miss_flag    //Assumed false by default; Set on coherence miss
  False_Sharing_flag     //Assumed true by default; Reset on True sharing detection

MAIN
  ON Write access to A DO
    IF (cache miss AND B is present in INVALID or READ_ONLY state) THEN
      CALL SUB record_coherence_miss;
    IF (Coherence_Miss_flag == true AND False_Sharing_flag == true) THEN
      //True sharing if another core had read(consumer)/written(producer) to this word
      IF (Write_vector_of_others[A] == 1 OR Read_vector_of_others[A] == 1) THEN
        False_Sharing_Flag = false;
    Local_Write_vector[A] = 1;
    Local_Read_vector[A] = 0;

  ON Read access to A DO
    IF (cache miss AND B is present in INVALID state) THEN
      CALL SUB record_coherence_miss;
    IF (Coherence_Miss_flag == true AND False_Sharing_flag == true) THEN
      //True sharing if another core had written(producer) to this word
      IF (Write_vector_of_others[A] == 1) THEN False_Sharing_Flag = false;
    Local_Read_vector[A] = 1;

  ON Invalidation/Downgrade/Replacement request for B DO
    FOR each word address K in B DO
      IF (Local_Write_vector[K]==1) THEN
        Global_Writer_ID = c;
        Global_Reader_vector={0};
      IF (Local_Read_vector[K]==1) THEN
        Global_Reader_vector[c]=1;
    DONE
    IF (Coherence_Miss_flag == true) THEN
      Record Miss_PC and False_Sharing_flag; //Output to Profiler
END MAIN

SUB record_coherence_miss
  WAIT ON current sharers of B to update Global_Writer_ID and Global_Reader_vector
  Set Coherence_Miss_flag to true;
  Set False_Sharing_flag to true; //Reset on observing first true sharing on the block
  Record the current PC into Miss_PC;
  Write_vector_of_others = Read_vector_of_others = {0};
  FOR each word address K in B DO
    //Writes clear Global_Reader_vector. Global_Reader_vector[c]=1 denotes NO intervening write
    IF (Global_Writer_ID!=c AND Global_Reader_vector[c]!=1) THEN Write_vector_of_others[K] = 1;
    IF (a bit other than c is set in Global_Reader_vector) THEN Read_vector_of_others[K] = 1;
  DONE
END SUB
```

Figure 5. False Sharing Detection Algorithm (o-FSD).

the information from the central repository and updates its local state to perform coherence miss classification.

4. A coherence miss may be classified as a true or false sharing miss many cycles after the instruction causing the miss has retired from the core's pipeline (Section 3.2). This delay in classification makes it more

difficult and costly to attribute false and true sharing misses to particular instructions, especially if performance counters support precise exceptions for events, such as PEBS (Precise Event Based Sampling) mechanism available in recent Intel processors [2]. For accurate attribution, the PC of the instruction that caused a coherence miss must be kept until the miss is eventually classified. This increases the cost of the classification mechanism, and also makes it more difficult for profilers to extract other information about the miss (e.g., what was the data address or value) [2].

To provide a realistically implementable scheme for online classification of coherence misses, we need to overcome some (preferably all) of the above problems, while sacrificing as little classification accuracy as possible. When choosing which aspects of classification accuracy to sacrifice, we keep in mind the primary purpose of our classification mechanism: giving the programmer an idea of how much performance is affected by true and false sharing cache misses, and pinpointing the instructions (and from there, lines of code) that suffer most of these misses. Armed with this, the programmer should be able to make an informed decision about how best to reduce the performance impact of these misses.

4. Implementation of False Sharing Detectors

In this section, we explore two different hardware implementations for False Sharing Detectors. First, we describe how o-FSD can be realized in hardware with reduced global state. Then, we show a practical mechanism that can classify coherence misses at almost zero cost and discuss the drawbacks associated with such a scheme. Finally, we briefly talk about issues relating to coherence miss classification for non-primary caches.

4.1. Hardware implementation of o-FSD (HW1)

Section 3.3 enumerated a number of issues that limit o-FSD from being implemented in real machines. The key to overcoming these problems is to reduce the amounts of both global and local state. In particular, o-FSD is very expensive because it maintains the global state for the entire memory throughout the program execution. The amount of global state grows in proportion to the number of cores and the memory overhead increases as a percentage of data memory with the number of cores. This lack of scalability makes this solution impractical for many-core systems.

In order to reduce the amount of global state, we maintain information only for cache blocks in the on-die caches. This can lead to misclassification of coherence misses to blocks that are evicted from all of the on-die caches between an invalidation/downgrade and the coherence miss. Evictions of such lines are highly likely to be a small fraction of the evictions, and thus this event should be quite uncommon. The state can be kept with the lowest level shared cache, for systems that have one. For chips with only private caches, and a separate mechanism for maintaining coherence such as a directory, the state can be kept with the coherence state.

This implementation still preserves the local state needed for every cache block. Also, the coherence protocol should be changed to carry the extra information to update the global state. However, since global state updates happen on cache block invalidations and coherence misses, the information to be relayed between local cache and the shared cache can be piggybacked on existing coherence messages. Further optimizations such as sampling a specific set of cache blocks or incorporating machine learning techniques are possible. Such tools might reduce cost and still maintain an acceptable accuracy for performance debugging. However, in this paper, we do not study such optimizations.

4.2. Local False Sharing Detector (HW2)

A second False Sharing Detector implementation *infers* false and true sharing locally by comparing the stale value of the data address suffering the coherence miss with the incoming value. If the data value has changed, then true sharing is detected because another core has produced a new value that this core currently consumes. Otherwise, the coherence miss is attributed to false sharing. This design does not maintain any state and requires only a trivial change in the cache controller to perform data comparison. However, it has the potential to overestimate false sharing misses in a program because early classification would ignore true sharing access on a block that might happen much after the point of coherence miss (as discussed in Section 2.1). This technique of detecting false sharing through data comparison was used by Coherence Decoupling [10] to save cache access latency due to false sharing misses.

Apart from the potential of overestimation, this implementation could misclassify in two situations: 1) Silent stores [11] do not change the data value and hence, it is impossible to detect true sharing in such situations. However, classifying silent stores itself is a murky area. Whether they contribute to true or false sharing depends on specific situations. For example, lock variables have an initial value of zero. A core grabs a lock by setting it to one. Later when the lock is freed, it deposits a value of zero back. An external core does not see change in lock value and effectively sees a value of zero before and after the core operated on the lock. Even though lock sharing is a form of true sharing, detecting false sharing through data comparison would miss this effect. At the same time, other silent writes that simply do not communicate any new value can be eliminated through algorithmic changes. 2) While readers would be able to detect change in data value due to an external write, writers do not detect true sharing as readers do not modify data. As long as coherence misses are classified on the consumer side correctly, the programmer might still get a sufficient picture of false sharing effects in the program. Note that, our first hard-

ware implementation (HW1) does not have the above problems, as it tracks the reader and writer information directly.

4.3. Non-primary Private Caches

Local state is relatively easy to update and check in a primary (L1) cache where all memory accesses are visible to the cache controller. In lower-level caches, only cache line addresses are visible. As a result, in systems where multiple levels of cache may be involved in coherence (e.g., with private L2 caches), information from a private non-primary (L2) cache should be passed to the primary cache (L1) when it suffers a cache miss. The L1 cache then keeps track of the flags that track coherence miss and false sharing for the L2 cache and forwards the value of these flags back to L2 when the block is replaced from the L1 cache.

5 Related Work

True and false sharing misses have been defined by Torrellas et al. [17], Eggers et al. [8], and Dubois et al. [7]. Foglia et al. [9] proposed an algorithm for coherence miss classification by maintaining false sharing transaction records on the writer side to detect keep track of readers. Each of them also describe an off-line classification algorithm. In contrast to these schemes and definitions, the mechanisms we describe in this paper are designed to be implemented in real hardware for integration with existing on-line performance debugging infrastructures. Tools such as MemSpy [12], SIGMA [6] and SM-prof [4] are simulation-based performance debuggers to study memory bottlenecks.

Coherence Decoupling [10] uses local data comparison to detect false sharing. It speculatively reads values from the invalid cache lines to hide latency of a cache miss caused by false sharing. It then uses the incoming (coherent) data values to verify successful value speculation. If the values differ (true sharing of data between the cores), recovery action is triggered to recover from misspeculation.

Numerous research proposals have been made for improving the performance counter infrastructure [16], attribution of performance-related events to particular instructions [5], and for sampling and processing of profiling data [2, 13, 14, 19]. Our cache miss classification mechanisms are synergistic with improvements in performance counters, sampling, and profiling infrastructure. Our mechanisms provide on-line identification of specific types of cache misses, and this identification can be used to drive performance counters, attributed to particular instructions, and processed further to gain more insight into program behavior and performance. The better the profiling infrastructure, the more beneficial the results of our classification are to the programmer. Conversely, our scheme enhances the value of a profiling infrastructure by providing additional event types that can be profiled.

Benchmark	Input	Benchmark	Input
Barnes	16K	Cholesky	tk29.0
FFT	64K	FMM	16K
LU	512x512	Ocean	258x258
Radiosity	-room	Radix	256K
Raytrace	car	Volrend	head
Water-sp	512	Water-n2	512

Table 1. Splash-2 benchmarks and their inputs.

Benchmark	Input
Blackscholes	16k options
Bodytrack	4 cameras, 2 frames, 2000 particles, 5 layers
Facesim	80598 particles, 1 frame
Fluidanimate	100000 particles, 5 frames
Swaptions	32 swaptions, 10000 simulations

Table 2. PARSEC benchmarks and inputs.

6. Evaluation Setup

We evaluate o-FSD and our hardware implementations using SESC [15], a cycle-accurate execution-driven simulator. The configuration we model is a 64-core chip multiprocessor. Each core is a 2.93GHz, four-issue, out-of-order processor with a 32KB, 4-way set-associative private L1 cache and a 256KB, 16-way set-associative private L2 cache. All cores share an 8MB, 32-way L3 cache, and the MESI protocol is used to keep L2 caches coherent. The block size is 64 bytes in all caches.

We use two sets of benchmarks for our evaluation: the Splash-2 benchmark suite [18] (Table 1) and a subset of benchmarks from the PARSEC-1.0 [3] benchmark suite (Table 2). Both benchmark suites are highly scalable and thoroughly tuned. Therefore, our experiments show the accuracy of our hardware mechanisms with respect to o-FSD in these applications, but we do not expect to find actual scalability problems. We simultaneously perform attribution to individual program counters to measure how many true or false sharing misses were contributed by a particular static instruction. This enables us to achieve deeper insight into whether the top set of offender instructions in a particular implementation match with the top offender instructions reported by o-FSD. It helps determine whether the programmer would still get a meaningful picture about false sharing patterns in the program.

We note that performance overheads could arise out of two scenarios: 1) when primary level cache needs to update its local read/write vectors for every access, there is a 0.41% additional latency for 32 KB L1 caches used in our experiments and 2) when the shared bus becomes saturated with extra traffic needed for reading and updating information in the central repository or global state. We did not find any instances of bus saturation in our experiments. All other latencies for local state updates can be hidden by looking up state in parallel with data accesses and/or updating state after data access completes. Therefore, we do not perform any latency related experiments in our evaluation.

7. Evaluation

In this section, we examine the classification accuracy of our hardware implementations with respect to o-FSD. We begin by examining HW1, which is more accurate (with respect to o-FSD) and expensive practical scheme that maintains global and local states for all cache blocks. Then, we compare HW2 against o-FSD that detects false sharing misses locally through data comparison and examine how much accuracy is offered by this scheme.

To examine how the approximations in hardware implementations affect their accuracy, we measure the percentage of coherence misses that are correctly classified by the particular scheme. We refer to this metric simply as 'accuracy'.

Figure 8 shows the accuracy results for Splash-2 and PARSEC benchmarks respectively. Each benchmark is evaluated for our two different hardware implementations. The left bar shows the accuracy results for HW1 and the right bar shows the accuracy results for HW2 scheme.

HW1 achieves perfect accuracy in all of our benchmarks expect ocean where a small percentage ($<$1%) of coherence miss classifications disagree with o-FSD. This inaccuracy results out of loss of global state that happens over very long time intervals.

HW2 offers relatively good accuracy in some benchmarks (fft, lu, ocean, blackscholes) but performs poorly in other benchmarks with a minimum accuracy of 35%(radiosity). Even for benchmarks where accuracy is lower, we find that top ten static instructions seen as causing false sharing misses by HW2 are similar to the top ten static instructions seen by o-FSD, although they differ significantly in the number of false sharing misses reported for the corresponding offenders. A large percentage of inaccuracy observed in HW2 comes from not adopting programmer-centric definition of false sharing as described in Section 2.1. HW2 performs classification at the time of coherence miss.

Additionally, write misses in Radix, and to a lesser extent Barnes and FMM are misclassified by HW2 due to lack of information about true sharing reads performed by other cores (See Section 4.2). Hence, the writer cores incorrectly identify such misses as false sharing misses. Even with this limitation, HW2 classifies writes correctly in many applications mainly because a core that reads from a shared variable often writes to it as well, allowing other writers to identify true sharing between their own and other's writes even though other's reads are unknown. In Radix, however, most values are read by a thread without being updated, so when a write does occur and causes a coherence (upgrade) miss, that miss is mistakenly classified as a false sharing miss. The same behavior occurs, albeit less frequently, in Barnes, and FMM. However, our results for all benchmarks (including Radix, Barnes, and FMM) show that read misses tend to be a dominant form of coherence misses. For example, in Radix it is clear that the dominant problem is false sharing read misses so the misclassification of write misses is not going to lead the programmer into trying to solve a non-extant write-false sharing problem.

We also investigated the cause of read miss misclassification in Radiosity. When misses are attributed to individual static instructions, we find that a small number of instructions account for a very high percentage of true sharing misses. A significant fraction (but not the majority) of these misses get misclassified as false sharing misses in HW2 scheme due to idempotent writes (See Section 4.2) observed by a core with respect to writes performed by other cores. Radiosity uses dynamic scheduling with task queues. Nearly all misclassified misses (and most of the correctly classified ones) in this application occur in the scheduling code. The idempotent writes occur in the following manner: when the task queue is empty, threads that have completed their work repeatedly check it to see if a new task has been inserted. When a working thread does such an insertion (and invalidates cached copies kept by the waiting cores), some of these waiting threads read the new values of the queue head and related counters. These values have changed, so these read misses are correctly classified as true sharing misses. One of these threads then grabs the task from the queue and returns it to an empty state (invalidating the other's cached copies), so subsequent read misses by the other waiting threads are again correctly classified. However, some of the waiting threads did not get a chance to examine the queue between the two changes (the insertion and subsequent removal of a task). Instead, when they check the queue, it is already empty again, and observe the same values they had previously observed (empty queue). As a result, the stale cached values and the incoming coherent values are equal and in HW2 scheme the read miss is incorrectly classified as a false sharing miss. In fact, two true-sharing writes have occurred, but the second undoes the effect of the first. Note that these misclassified read misses occur along with a large number of correctly classified true sharing read misses (when both written values are observed). As a result, the majority of the misses attributed to each instruction are still correctly classified, and the dominant behavior (frequent true sharing as threads spin-wait on empty task queues) is still clearly identified by HW2.

8 Conclusions

Performance debugging of parallel applications is extremely challenging, but achieving good parallel performance is critical to justify the additional expense of parallel architectures. It is currently very difficult for programmers to diagnose (and therefore fix) a common scalability problem: coherence misses. Programmers must not only find the source of these misses, but in order to reduce them, must know whether the misses are due to true or false sharing.

We examine two different hardware schemes for on-the-fly detection and classification of coherence misses, and provide insights into the tradeoff between accuracy and cost. We

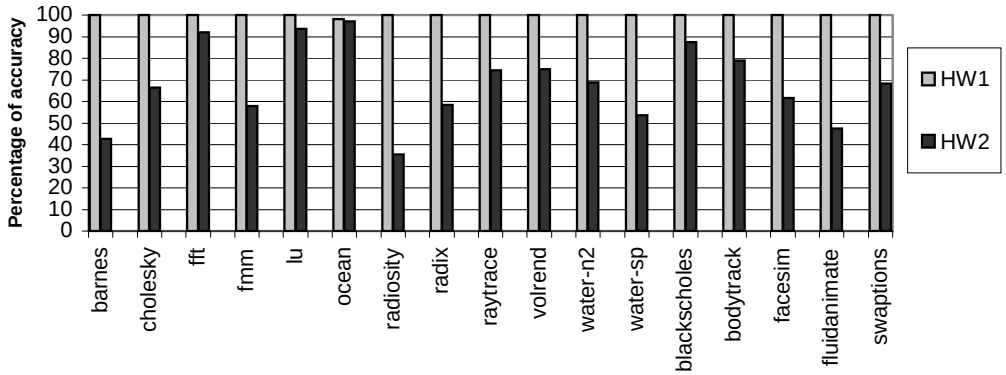

Figure 8. Accuracy of hardware false sharing detection schemes. For each benchmark, the left bar shows the accuracy results for HW1 implementation and the right bar shows the results for HW2 implementation.

evaluate our schemes on SPLASH-2 and PARSEC benchmarks and find that the simplest scheme, which keeps no state at all, achieves reasonable accuracy in some benchmarks but yields poor results in others. Conversely, we find that a comprehensive and expensive scheme offers nearly perfect accuracy.

References

[1] *Intel 64 and IA-32 Architectures Software Developer s Manual Volume 3B: System Programming Guide, Part 2*, Chapter 18: Debugging and Performance Monitoring. Intel Corporation, 2007.

[2] A. Alexandrov, S. Bratanov, J. Fedorova, D. Levinthal, I. Lopatin, and D. Ryabtsev. Parallelization Made Easier with Intel Performance-Tuning Utility. *Intel Technology Journal*, 2007.

[3] C. Bienia, S. Kumar, J. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. *Princeton University Tech. Rep. TR-811-08*, 2008.

[4] M. Brorsson. Sm-prof: a tool to visualise and find cache coherence performance bottlenecks in multiprocessor programs. In *SIGMETRICS '95/PERFORMANCE '95: 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 178–187, New York, NY, USA, 1995. ACM.

[5] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos. Profileme : Hardware support for instruction-level profiling on out-of-order processors. In *Intl. Symp. on Microarchitecture*, pages 292–302, 1997.

[6] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia. Sigma: a simulator infrastructure to guide memory analysis. In *Supercomputing '02: 2002 ACM/IEEE conference on Supercomputing*, pages 1–13, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[7] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom. The Detection and Elimination of Useless Misses in Multiprocessors. *USC Tech. Rep. No. CENG 93-02*, 1993.

[8] S. Eggers and T. Jeremiassen. Eliminating false sharing. In *Intl. Conf. on Parallel Processing*, pages 377–381, 1991.

[9] P. Foglia. An algorithm for the classification of coherence related overhead in shared-bus shared-memory multiprocessors. In *IEEE TCCA Newsletter*, Los Alamitos, CA, USA, 2001. IEEE Computer Society Press.

[10] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: making use of incoherence. In *ASPLOS*, pages 97–106, 2004.

[11] K. Lepak and M. Lipasti. Temporally Silent Stores. In *10th Intl. Conf. on Arch. Support for Prog. Lang. and Operating Sys.*, 2002.

[12] M. Martonosi, A. Gupta, and T. Anderson. Memspy: analyzing memory system bottlenecks in programs. In *SIGMETRICS '92/PERFORMANCE '92: 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 1–12, New York, NY, USA, 1992. ACM.

[13] H. Mousa and C. Krintz. Hps: Hybrid profiling support. In *PACT '05: 14th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 38–50, Washington, DC, USA, 2005. IEEE Computer Society.

[14] P. Nagpurkar, H. Mousa, C. Krintz, and T. Sherwood. Efficient remote profiling for resource-constrained devices. *ACM Trans. Archit. Code Optim.*, 3(1):35–66, 2006.

[15] J. Renau et al. SESC. *http://sesc.sourceforge.net*, 2006.

[16] S. S. Sastry, R. Bodík, and J. E. Smith. Rapid profiling via stratified sampling. In *ISCA '01: 28th annual international symposium on Computer architecture*, pages 278–289, New York, NY, USA, 2001. ACM Press.

[17] J. Torrellas, M. J. Lam, and J. L. Hennessy. Shared data placement optimizations to reduce multiprocessor cache misses. In *Intl. Conf. on Parallel Processing*, pages 266–270, 1990.

[18] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Intl. Symp. on Computer Architecture*, 1995.

[19] C. B. Zilles and G. S. Sohi. A programmable co-processor for profiling. In *HPCA '01: 7th Intl. Symp. on High-Performance Computer Architecture*, page 241, Washington, DC, USA, 2001. IEEE Computer Society.