DeFT: Design Space Exploration for On-the-Fly Detection of Coherence Misses

GURU VENKATARAMANI, The George Washington University
CHRISTOPHER J. HUGHES, Intel Corporation
SANJEEV KUMAR, Facebook Inc.
MILOS PRVULOVIC, Georgia Institute of Technology

8

While multicore processors promise large performance benefits for parallel applications, writing these applications is notoriously difficult. Tuning a parallel application to achieve good performance, also known as performance debugging, is often more challenging than debugging the application for correctness. Parallel programs have many performance-related issues that are not seen in sequential programs. An increase in cache misses is one of the biggest challenges that programmers face. To minimize these misses, programmers must not only identify the source of the extra misses, but also perform the tricky task of determining if the misses are caused by interthread communication (i.e., coherence misses) and if so, whether they are caused by true or false sharing (since the solutions for these two are quite different).

In this article, we propose a new programmer-centric definition of false sharing misses and describe our novel algorithm to perform coherence miss classification. We contrast our approach with existing data-centric definitions of false sharing. A straightforward implementation of our algorithm is too expensive to be incorporated in real hardware. Therefore, we explore the design space for low-cost hardware support that can classify coherence misses on-the-fly into true and false sharing misses, allowing existing performance counters and profiling tools to expose and attribute them. We find that our approximate schemes achieve good accuracy at only a fraction of the cost of the ideal scheme. Additionally, we demonstrate the usefulness of our work in a case study involving a real application.

Categories and Subject Descriptors: C.1.0 [**Processor Architectures**]: General

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Performance debugging, false sharing, multicore processors, coherence misses

ACM Reference Format:
Venkataramani, G., Hughes, C. J., Kumar, S., and Prvulovic, M. 2011. DeFT: Design space exploration for on-the-fly detection of coherence misses. ACM Trans. Architec. Code Optim. 8, 2, Article 8 (July 2011), 27 pages.
DOI = 10.1145/1970386.1970389 http://doi.acm.org/10.1145/1970386.1970389

This material is based on work supported by the National Science Foundation under Grants Nos. 0447783, 0541080, 0993470, by Semiconductor Research Corporation (SRC) under contract 2009-HJ-1977. G. Venkataramani is supported by 2010 ORAU Ralph E. Powe Junior Faculty Enhancement Award and GWU Dilthey Faculty Fellowship.
Authors' addresses: G. Venkataramani, Department of Electrical and Computer Engineering, George Washington University, 801 22nd St. NW Suite 624D, Washington, DC 20052; email: guruv@gwu.edu; C. J. Hughes, Intel Corporation; S. Kumar, Facebook, Inc.; M. Prvulovic, Georgia Institute of Technology.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2011 ACM 1544-3566/2011/07-ART8 $10.00
DOI 10.1145/1970386.1970389 http://doi.acm.org/10.1145/1970386.1970389

1. INTRODUCTION

Multicore processors provide the potential for a huge increase in general-purpose processor performance. Tapping this potential requires efficient use of the multiple cores on the chip; that is, applications must be written so that their performance scales with the number of available cores. This is often extremely challenging, even when the underlying application is amenable to parallelization, because multithreaded (parallel) applications can suffer performance problems that either do not exist in serial execution or that are aggravated by parallel execution.

One important class of such performance limiters consists of *coherence misses*, which are caused by communication of data between cores, either intentionally (true sharing) or unintentionally (false sharing). Since coherence misses do not occur in uniprocessor execution, 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 are left to *guess* whether an increase in misses is caused by coherence and whether such misses are caused by true or by false sharing. Further, most programmers do not have an in-depth understanding of computer architecture, which leads them to interpret "excessive cache misses" as "data does not fit in cache." Even those programmers who are aware of the concept of coherence misses are still often surprised by false sharing misses; without sufficient knowledge of the underlying hardware, the presence of these misses is mysterious and even frustrating.[1]

Not understanding the underlying reason for excessive cache misses would not be an issue if the same set of techniques could be used to alleviate all types of cache misses. Unfortunately, the techniques for addressing different types of cache misses are fundamentally different. Noncoherence misses in long-running applications and on typical hardware are typically dominated by capacity misses, which are usually alleviated by reducing the working set size using blocking transformations, more space-efficient data structures, etc. True sharing misses are typically reduced by changing the assignment of tasks to threads, or by using a different parallel algorithm. Finally, false sharing is often easily addressed by separating affected data in memory, for example, by adding padding.

Processor vendors understand that making processors with increased raw performance is not sufficient: customers must see a real performance difference. This requires that software vendors write their applications to take advantage of the increased raw performance. To aid programmers in finding and eliminating performance bottlenecks in their code (i.e., performance debugging), most processor vendors currently invest some of the chip resources into support for performance counters [Intel 2007].

Unfortunately, existing solutions are either insufficient or inefficient for accurately classifying coherence misses. Current performance counter infrastructures can inform programmers which points in a program are generating cache misses, but do not specifically count coherence misses, which often leads programmers to attempt time-consuming but futile (or even harmful) program transformations.

To overcome this problem, a profiling infrastructure should detect and breakdown misses into noncoherence, true sharing, and false sharing misses, which can be separately attributed to particular points in the code and reported to the programmer with minimal impact on original program behavior. With such an infrastructure, the developer can focus on solving the performance problem rather than figuring out which hardware mechanism is causing it.

[1]Our anecdotal evidence from observing inexperienced developers (senior-year computer science students at Georgia Tech) suggests that many have not internalized even that false sharing exists as a concept, so they are completely perplexed when their parallel code performs poorly due to these misses.

Such profiling infrastructure can be implemented either as a software tool or as an extension to the existing profiling counter infrastructure. A software tool would either simulate or instrument an application to capture the memory behavior. Our I-FSD algortihm (described in Appendix A) can provide such functionality. However, a software-only false sharing analysis tool would have some inherent drawbacks, in addition to requiring programmers to add yet another tool to their repertoire. In particular, the tool may perturb the behavior of the software it is analyzing in at least two ways: (1) Instructions added by the tools could create additional memory accesses and interfere with normal memory access patterns. (2) False sharing may depend on the timing of different threads execution, which can be perturbed by software tools. Also, a software false sharing detection tool will cause a large performance degradation (e.g., MemSpy [Martonosi et al. 1992], a software tool of similar complexity, gives a 57.9x slowdown). With this in mind, in this article, we also provide the intellectual framework and hardware support that is needed to extend existing performance counter infrastructure with cache miss classification.

The main contributions of this article are as follows.

(1) We provide the first *programmer-centric* definition of true and false sharing for invalidation-based cache-coherent machines, along with a novel algorithm to classify coherence misses. We then discuss the relationship between the new programmer-centric definition and prior dataflow-centric definitions. Unlike prior definitions, which focus on whether cache misses are necessary in the dataflow sense (i.e., is it needed to transfer data from a producer thread to a consumer thread), our definition focuses on whether or not the cache miss would occur if different data items were in different cache blocks. Intuitively, prior definitions ask "Is this communication needed on every possible machine to obey the same true dependence (dataflow) between threads?" In contrast, our definition asks "Will a miss still occur on this particular machine if variables in memory are allocated differently?"
(2) Although there have been several proposals for cache miss classification offline or in architectural simulators [Dubois et al. 1993; Mattson et al. 1970; Torrellas et al. 1990], to our knowledge, this work is the first to propose classification of cache misses on-the-fly in real hardware to drive performance counters and hardware-assisted profiling.
(3) We explore the design space for different implementations of cache miss classifiers (which we call as DeFT) based on our programmer-centric definition. This includes a detailed cost-usefulness trade-off analysis and evaluation of several different approximations that sacrifice classification accuracy to reduce hardware cost.

2. FALSE SHARING AND ITS IMPLICATIONS ON SCALABILITY

In cache-coherent Chip Multiprocessors (CMP), data sharing between threads primarily manifests 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, which is intuitive to most programmers. One example is when a consumer (reader) of the data must suffer a cache miss to obtain the updated version of the data from the producer (writer). Scalability issues stemming from true sharing misses can typically be addressed only by changing the algorithm, the distribution of work among cores, or synchronization and timing.

In contrast, false sharing misses are an artifact of interaction between data placement and cache blocks. As an example, consider a block containing two data items where each data item is (exclusively) used by a different core. Every coherence miss

```
// 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;
...
```

Fig. 1. 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]++;
...
```

Fig. 2. A parallel histogram computation illustrating false sharing on an indirectly accessed array. Locking of counter_array elements is omitted for brevity.

that happens on this cache block is a result of false sharing between these cores. Scalability issues arising from false sharing are often alleviated by changing alignment or by adding padding between affected data items.

2.1. Real-World Examples of False Sharing as a Scalability Limiter

We provide examples, taken from code written by experienced programmers, to illustrate some common situations where false sharing can occur and have a (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 1. There is no true sharing in this code, as 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 was encountered in real-world Web search, fluid simulation, and human body tracking applications. A common fix is to add padding around each counter. As a real-world example, we use *facesim* from the PARSEC-1.0 benchmark suite. The benchmark's authors spent multiple days to eventually identify that false sharing from this loop is 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 programmers. A case study of applying our proposed support to this code is presented in Section 3.5.

Our second example, shown in Figure 2, involves an indirectly accessed data array. It 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 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;
}
...
```

Fig. 3. A red-black Gauss-Seidel-style array update showing false sharing on a finely partitioned array.

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 happens 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 is 15 times more likely than true sharing. A common fix is to either add padding around each element (which addresses only false sharing) or to use privatization[2] to reduce both true and false sharing. This reinforces the importance of knowing which types of misses are happening; if most are due to false sharing, padding should be used, but if true sharing contributes significantly to coherence misses, then privatization is needed. 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, mostly due to false sharing. With privatization, the benchmark achieves near-linear scalability.

Our final example of false sharing involves finely partitioned arrays, such as those in red-black Gauss-Seidel computation shown in Figure 3, which is again taken from real-world code. 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). The red-black approach avoids synchronization on each element by treating the array as a checkerboard with 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 of a different color than the cell being updated, and therefore not updated during the current pass even if they belong to another thread's partition.

Both true and false sharing occurs in this example. The first time a thread accesses a cache block across a partition boundary, it incurs a true sharing miss because it is reading data written by another thread during the previous pass. However, that other thread is actively updating elements in the same cache block, which 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 was encountered in real-world 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 preceding 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

[2]Privatization involves using a separate (private) array for each thread, then merging partial results at the end.

loss in spatial locality. Grouping multiple rows together to be processed by the same thread, and using two separate arrays for red and black cells improved the scaling to almost 6x.

3. DETECTION OF TRUE AND FALSE SHARING MISSES

In this section, we first describe the relatively simple mechanism for distinguishing coherence misses from noncoherence misses, then discuss an Ideal False Sharing Detector (I-FSD) mechanism that further classifies coherence misses into those caused by false sharing and those caused by true sharing. We then contrast our I-FSD algorithm with the algorithm proposed by Dubois et al. [1993] and show the key differences between them.

3.1. Identification of Coherence Misses

Coherence misses can be distinguished from other (cold, capacity, or conflict) misses by checking if the cache block is already present in the cache. For noncoherence 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 identified as a coherence miss if a matching tag is found but the block does not have an appropriate state (for MSI coherence protocol, finding a cache block in Invalid state for read operations, or Shared/Invalid states for write operations); conversely, the miss is identified as a noncoherence miss if no block with a matching tag is found. This technique has a few corner cases that are further elaborated in Section 4.5, but is still highly accurate.

3.2. Need for a Programmer-Centric Definition of False Sharing

We define false and true sharing misses based on whether the miss can be avoided if the data item(s) involved are placed in different cache block(s). In contrast, prior works [Dubois et al. 1993; Torrellas et al. 1990] define false and true sharing misses according to whether the miss is necessary, that is, whether the miss is needed to communicate a new value regardless of the underlying hardware. Note that this is not meant to imply that prior definitions are incorrect. Instead, the different definitions are driven by different needs. Prior definitions were derived for the purpose of comparing the behavior of different coherence mechanisms and algorithms; our aim is to help programmers and/or compilers decide how to modify code to improve performance on a real cache-coherent machine with an invalidation-based protocol.

3.3. Ideal False Sharing Detector (I-FSD) Based on Programmer-Centric Definition

We define true and false sharing using the notion of "overlapping" accesses by different cores to the same line. When core A incurs a coherence miss on a line, it is a true sharing miss if and only if there is at least some "overlap" between the parts of the line core A accesses and the parts of the line that were accessed by other cores since the line was invalidated or downgraded in core A's cache. "Overlap" exists when other cores have accessed the same memory location in a way that requires coherence action when core A accesses it. Thus, a read access from the core A has overlap with a write from another core if the address range being read intersects the address range written. Coherence actions are necessary in this case to bring the new value to the core A. Similarly, a write access from core A has overlap with a read or write access from another core if their address ranges intersect. Coherence actions are necessary in this case to invalidate the stale version of the data in the caches of other cores. As we will discuss in Section 3.4, this coherence-action-oriented overlap definition is the key

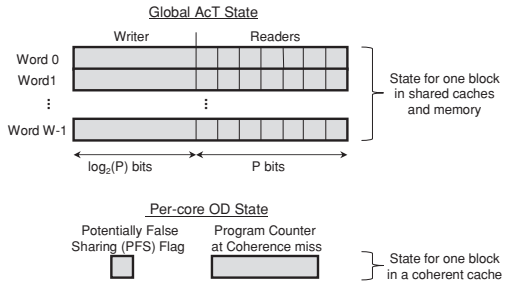


Fig. 4. State maintained and used by our I-FSD coherence miss classification algorithm.

difference between our programmer-centric definition and prior definitions of true and false sharing.

Our definition considers overlap between accesses from other cores and all accesses from core *A* between the miss and when the line is replaced, invalidated, downgraded, or upgraded.[3] This means that while the line is in the cache, if any access has overlap with other cores, the coherence miss that brought the line into the cache (or upgraded it) is a true sharing miss. This is needed because it is possible that the access that triggers the miss may not be involved in true sharing, but subsequent accesses to different parts of the same block may be. Because the miss being considered has brought the entire block into the cache, these subsequent true sharing accesses do not result in coherence actions. However, if the original miss is treated as a false sharing miss and the code is modified to avoid it (e.g., by separating the data items into different cache blocks), the true sharing would still result in a miss. In light of this, a coherence miss that is involved in both true and false sharing is defined to be a true sharing miss. This is similar to prior definitions (e.g., Dubois et al. [1993]).

Appendix A presents an algorithm that uses the preceding definition to distinguish between true and false sharing misses. This Ideal False Sharing Detector (I-FSD) has two main mechanisms: an Access Tracker (AcT) and an Overlap Detector (OD). The AcT mechanism maintains the information about past accesses by different cores that will be needed to classify future coherence misses. After a coherence miss occurs, the OD mechanism determines whether that miss was caused by true or false sharing.

I-FSD keeps the state shown in Figure 4 for its AcT and OD mechanisms. For each word in main memory and shared caches, I-FSD tracks which core was the last to write to that word (*Writer* field) and which cores have read that word since it was last written (*Readers* bit-vector). We refer to this state as the global AcT state. For each private cache block, I-FSD keeps a Potentially False Sharing (PFS) bit. It sets this bit at the time of a coherence miss to indicate that this line might have been brought into the cache (or upgraded) because of a false sharing access. On every access to a line with the PFS bit set, I-FSD uses the global AcT state to check if the access has overlap with another core; as described earlier, this means the coherence miss is a true sharing miss. When a line with the PFS bit set is invalidated, downgraded, upgraded, or replaced, I-FSD classifies the miss that brought the line in (or upgraded it) as a false sharing miss. For each private cache block, I-FSD also keeps a Program Counter (PC) field to record the static instruction that suffered a coherence miss on it for proper attribution once I-FSD finally determines if the miss was from true or false sharing.

[3]We treat upgrade misses (i.e., writes to lines in shared state) as coherence misses. If the line suffering an upgrade miss previously suffered a different coherence miss, we classify the previous miss as true or false sharing based on accesses up to the upgrade.

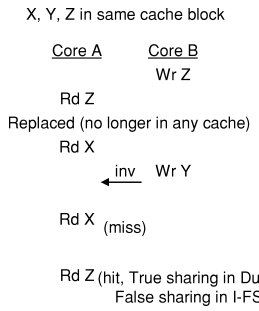


Fig. 5. Accurate classification requires us to keep access information even for blocks that are no longer in any cache.

3.4. Comparison with Dubois et al.'s Scheme

Dubois et al. [1993] propose algorithms to classify coherence misses for invalidation- and update-based cache coherence protocols. Since our I-FSD algorithm is intended for real invalidation-based coherence protocols that are implemented in modern multicore processors, we contrast it with the invalidate-based algorithm described by Dubois et al. [1993], which maintains a *stale bit* for every cache word. This stale bit is maintained for every word in the private-level caches and is used to track whether a new value was produced by any core to the word and whether the value was consumed by this core. When a write happens on a word, the caches that currently have the word clear the stale bit to indicate that a new value is produced. A subsequent read of this word denotes consumption of a fresh value produced by a write and indicates true sharing. This also sets the per-core stale bit so any further reads from the same core are found to use a stale value. When the cache block is replaced, the information regarding stale bits are discarded and the bits are reset to track the producer-consumer patterns of the new incoming block.

Both Dubois et al. [1993] and our scheme perform delayed classification of coherence misses and detect any true sharing of data between accesses by different cores. The difference is in how true and false sharing are detected for each memory access. In Dubois et al.'s scheme, true sharing is determined as dataflow from producers to consumers, and all other sharing patterns are considered to be false sharing. In contrast, our scheme classifies coherence misses based on whether the miss would be eliminated if data items were placed in different cache blocks.

As a result, there are two main cases where our I-FSD and Dubois et al. differ in classifying a miss. First, write accesses that overwrite data read or written by other cores are assumed nonessential (false sharing) in Dubois et al. [1993], but are considered true sharing in our I-FSD. In a *single producer-multiple consumer* pattern (e.g., Gauss-Seidel code in Section 2.1) and *multiple producer-multiple consumer* pattern (e.g., histogram code in Section 2.1), coherence misses are highly likely to happen in multiple cores (the producer and the respective consumers). Dubois et al. may classify producer-side misses as false sharing and consumer-side misses as true sharing, which may underestimate the effect of true sharing in such situations and confuse the programmer. Our I-FSD algorithm classifies producer- and consumer-side misses consistently.

Second, the Dubois et al. [1993] algorithm clears stale bits on every cache block replacement. This results in loss of history about the "staleness of values" (information that a consumer had already read the value from the cache word). When the word is subsequently read by the core (after another miss), the stale bit is zero and hence, true sharing is indicated by Dubois et al. Figure 5 shows an example where the cache block

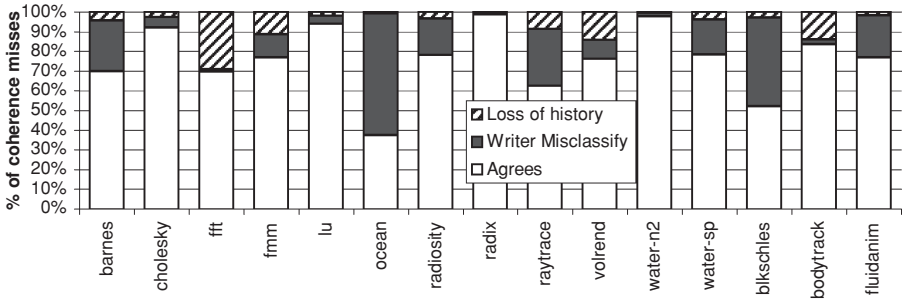


Fig. 6. Comparison between I-FSD and Dubois et al. Each bar has three portions- bottom portion shows the percentage agreement between the two schemes, and the middle and top portions show the percentage breakdown of disagreement between the two schemes.

having items X, Y, and Z has been replaced by both cores A and B. When core A suffers a coherence miss, Dubois et al. [1993] classifies "Read Z" as true sharing access because the information regarding the "staleness" of Z has been lost during cache block replacement. Our I-FSD algorithm uses the global AcT state to correctly classify the access as false sharing, which is the correct programmer-centric classification because the coherence miss would not happen if Y is separated from Z and X. We believe that these misclassifications in Dubois et al. stem from a desire to have more efficient implementation of coherence miss classification in a cache simulator. In contrast, our I-FSD is designed to provide a programmer-centric "ground-truth" classification against which we can evaluate approximate classification schemes. We note that any approximations that are crafted into I-FSD itself would jeopardize the soundness of the evaluation of further approximations in classification schemes.

Figure 6 shows results of experiments that compare coherence miss classification reported by our I-FSD and by Dubois et al. (configuration parameters used in our evaluation are shown in Section 6). The bottom portion of each bar shows the percentage of coherence miss classifications that Dubois et al. [1993] and I-FSD schemes agree on. In many benchmarks, the two schemes agree for at least 75% coherence misses except fft, ocean, raytrace, and blackscholes. The middle portion of each bar shows the percentage of coherence misses where our I-FSD identifies true sharing on the writer side, but Dubois et al. classifies the same misses as nonessential (false sharing). A large portion of such misclassifications occur in benchmarks such as ocean (62%), blackscholes (45%), raytrace (28%), and fluidanimate (21%). Finally, the top portion of each bar shows the percentage of coherence misses that Dubois et al. classifies as essential (true sharing) because the relevant stale bits were lost due to replacement, whereas our I-FSD algorithm uses its global AcT information to identify that these misses are actually caused by false sharing. This type of misclassification in Dubois et al. occurs relatively frequently in several benchmarks, such as fft (29%), volrend (14%), and bodytrack (14%).

3.5. Case Study: Facesim Benchmark

To illustrate how a cache miss classification scheme could be combined with existing performance counter infrastructure to provide insight for performance debugging, we present an example case study for the Facesim benchmark from the Parsec-1.0 suite [Bienia et al. 2008]. The code has been tested and fine-tuned for performance before release, but the original code that suffers false sharing is still present in the released code. We restore this early original code for this case study, and use I-FSD implemented in a simulator to classify coherence misses and attribute them to individual code addresses

Table I. Program Counters with False Sharing Misses in the Modified Facesim Benchmark

Program Counter	False Sharing Count
4cc084	9921186
4cc074	7803903
53eb34	251457
53f2ac	214323
53f5d8	201885
...	...
Total	18854631

(static instructions). The results of this attribution (Table I) point to two instructions, `4cc084` and `4cc074`, which are responsible for 24.5% of cache misses, and show that 94% of misses on these instructions are caused by false sharing. We use the *addr2line* utility from the GCC toolchain to identify source-code lines for these instructions. We find that they point to lines inside the loop where accumulators *rho_new* and *supernorm* are being updated in the *One_Newton_Step_Toward_Steady_State_CG_Helper_II()* function. We change the loop to accumulate the results locally into *local_new_rho* and *local_supernorm* and update *new_rho* and *supernorm* after the loop is complete. This change effectively restores the released version of the code, and it improves performance scaling from a 4x parallel speedup on a real 8-core machine to near-linear (almost 8x) speedup. Further details on scalability of this benchmark can be found in Bienia et al. [2008].

4. DESIGN SPACE EXPLORATION FOR PRACTICAL FALSE SHARING DETECTORS

Our ideal I-FSD scheme described in Section 3.3 achieves its accuracy by keeping global and per-core state for every word in memory which results in enormous space overheads. Also, a lookup of the word's global AcT State entry is needed on each access, making the scheme impractical for hardware implementation even if its hardware cost can otherwise be tolerated. Therefore, we first describe how our scheme can be adapted (without loss in accuracy) for more cache-friendly operation.

Further, to reduce memory space overheads, a reasonable approach would be to bound the state while still maintaining fairly good accuracy in classifying relevant coherence misses. In Sections 4.2 and 4.3, we describe and examine a range of design choices that trade off accuracy for lower hardware costs. The key to overcoming implementation costs posed by I-FSD is to reduce the state used by both its AcT and its OD mechanisms. We explore several designs that have varying cost versus accuracy trade-offs shown in Figure 7. We perform experimental evaluation of these design points in Section 7.

4.1. I-FSD with Fewer Accesses to Global State

I-FSD scheme requires a lookup of word's global AcT state entry on every access. In this section, we show how I-FSD can be adapted (without loss in accuracy) for more cache-friendly operation. This adaptation consists of keeping access updates in the local cache and only propagating those updates to the global AcT State on replacements and coherence actions. Similarly, for OD we can avoid frequent checks of the global AcT State by collecting relevant access information at the time of the miss and caching this information in the local cache. The additional state needed for these modifications is shown in Figure 8.

To keep the global AcT State up-to-date, we record which words have been read and written by the local core using per-core *Own-Access*[4] bits. A read simply sets the

[4]Own-Access indicates to the core's cache that the word has been accessed by its own (local) core and doesn't imply any ownership (in the coherence sense of the word).

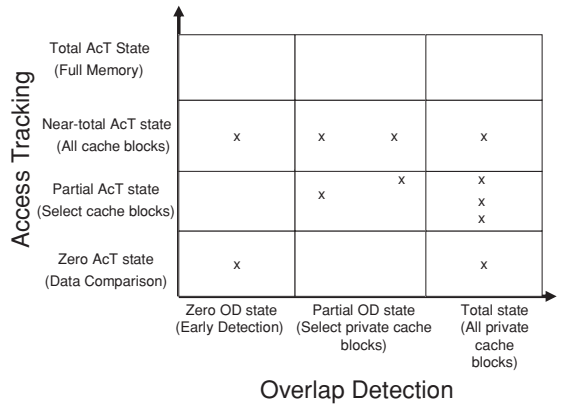


Fig. 7. Design space for coherence miss classification mechanisms. Design points marked "x" are examined in our evaluations.

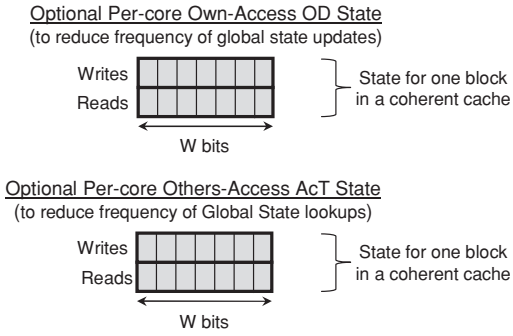


Fig. 8. Additional state to reduce frequency of global state accesses in our I-FSD coherence miss classification algorithm.

word's Own-Access *Read* bit. A write sets the *Write* bit and resets the *Read* bit. Each replacement, invalidation, upgrade, or downgrade received by the local cache results in sending the per-core Own-Access bits to the global AcT State, which is then updated accordingly.[5]

When a coherence miss occurs, the global AcT State of each word in that block can be examined (after it is updated as described previously) and the condensed *Others-Access* information can be sent to the requesting core.[6] With this per-core Others-Access information, the OD mechanism finds overlap when reading a word whose Others-Access Write bit is 1, or when writing a word where any (Read, Write, or both) Others-Access is 1.

While it may seem that some inaccuracy in classification can be caused by delaying updates of the global AcT State or by not updating per-core Others-Access bits after the miss being classified has occurred, such inaccuracy cannot occur. Delayed updates cannot be a source of inaccuracy because a coherence miss triggers updates to global

[5]If the *Write* bit is 1, the core's ID is placed into the Writers field and all Readers bits for that word are cleared; if a *Read* bit is set to 1, the AcT Readers bit corresponding to that core is set.

[6]The Others-Access Write bit for a word is set if the word's global Writer field points to another core. The Others-Access Read bit for a word is set if any Readers bit (except the one corresponding to the requesting core) for that word is set in the global AcT State.

AcT state, bringing that state sufficiently up-to-date to accurately classify that miss[7]. Similarly, the lack of updates to Others-Access state after a miss cannot result in incorrect classification: a write by another processor triggers an invalidation and immediate classification, so no stale Others-Access state is possible; a read by another processor either triggers a downgrade (and thus an update) or the local cache is only read-sharing the line and Reads bits in the Others-Access state are irrelevant (OD for reads only examines per-core Writer bits).

4.2. Approximate Access Tracking

The AcT scheme used in I-FSD maintains global AcT state for the entire memory throughout the program execution. We call this *Total AcT* in Figure 7. The number of bits needed for global AcT entries grows proportionally with both the number of cores and the size of the memory. The overhead (as a percentage of on-chip cache space) grows linearly with the number of cores; with 4 cores and 32-bit words, the storage overhead is 19%; with 32 cores, the storage overhead is 116%. Therefore, this solution is impractical in real hardware, but it is useful as a baseline for accuracy in our design space exploration.

The first simplification of AcT maintains AcT state only for cache blocks in the on-die caches. We call this *Near-Total AcT*, and its 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 (e.g., a directory), the classification state can be kept with the coherence state. This Near-Total AcT scheme can misclassify coherence misses due to "loss of history" between accesses (see example in Figure 5). Note that this loss of accuracy is not the same as the loss of "stale" bits in Dubois et al. [1993]; for example, using private L1 and shared L2 caches, Dubois et al. [1993] loses staleness information for any block that is not present in L1 caches, even if the block is still present in the (usually much larger) L2 cache. In contrast, our Near-Total AcT approximation only loses information for blocks that are no longer present in any on-chip cache (private L1 or the large shared L2). Furthermore, our Near-Total AcT is an approximate scheme and its accuracy will be compared to a fully accurate I-FSD scheme to find out exactly what the loss of accuracy is.

A more aggressive AcT simplification maintains only a limited pool of global AcT entries, and allocates entries from this pool on-demand. We call this *Partial AcT*. In this approach, an entry is allocated from the pool for a block when that block is either invalidated or downgraded, in anticipation of possible coherence misses on that block in the future. Subsequent accesses by various cores to the block are tracked and updated as described in the I-FSD algorithm. An AcT entry is freed when the block is no longer in any of the coherent caches (same as in Near-Total AcT), and also when the free pool is empty and an entry is needed for another block (using a replacement policy). Note that this approach cannot classify the very first coherence miss for a particular block but, because its allocation of AcT entries depends on seeing coherence actions, it actually spends the limited global AcT resources only on blocks that are actually involved in coherence. This Partial AcT approximation can lead to misclassification of coherence misses for any cache block whose AcT state is not tracked.

The most aggressive AcT simplification does not maintain any AcT state at all; it infers others-access information by comparing the stale value of the cache block with

[7]A write miss results in invalidations for all other sharers, whose Own-Access information is used to fully update Global State for the block. A read miss downgrades the previous writer (if any), whose Own-Access Write bits update the Global State's Writer fields for the block. Note that Reader bits are not needed to classify reads, and any writes will trigger full invalidation and bring Reader bits up-to-date.

incoming values.[8] If the data value has changed, then a write by another core is inferred and true sharing accesses is detected, otherwise, the access is classified as false sharing. We call this *Zero AcT*. This approximation suffers two additional sources of inaccuracy: (1) losing others-read information completely, which brings this scheme closer to Dubois et al. because reader-writer overlap is not detected as true sharing (but writer-writer patterns are still correctly classified), and (2) incomplete others-write information due to idempotent writes (e.g., lock variables before and after critical sections have identical values although during the critical section, the core sets and resets lock variables).

4.3. Approximate Overlap Detection

The OD implementation used in our I-FSD maintains per-core state for all blocks in coherent private caches. We call this *Total OD* in Figure 7, and we use this implementation as a baseline for OD accuracy.

The first simplification in OD we consider maintains a limited per-core pool of OD entries, and allocates an OD entry only when a coherence miss occurs[9], and frees it when the miss is eventually classified. We call this approach *Partial OD*, and it does not result in any misclassification of coherence misses. However, it may omit classification of coherence misses that occur when all entries in the per-core pool are in use. Still, Partial OD only omits miss classification when many other misses are being classified, so it is expected to still classify many coherence misses in spite of its lower cost.

The most aggressive simplification to OD does not maintain per-core state corresponding to OD for any cache block, and classifies a miss based only on the access that triggers it. We call this *Zero OD*, and note that it can overestimate the number false sharing misses by missing subsequent true sharing accesses to the same block (as discussed in Section 3.3).

Even though the preceding discussion appears to categorize Access Tracking and Overlap Detection into discrete design points, in reality, we get a continuous spectrum of design points along both axes, with Partial State providing a continuum between Total and Zero states along either axis. Ideally, a design point should be selected to minimize the hardware cost while still providing the programmer with enough accuracy to correctly diagnose performance bottlenecks.

4.4. Handling of Nonprimary Coherent Caches

Per-core state is relatively easy to update and check in a primary (L1) cache, where the actual address of each access is visible to the cache controller. In lower-level caches, only cache line addresses for misses and replacements in higher-level caches are visible. For systems where multiple levels of cache may be involved in coherence (e.g., with private L2 caches), information from a private nonprimary (L2) cache should be passed to the primary cache (L1) when it suffers a cache miss. The L1 cache then updates per-core AcT and OD state and forwards this information back to the L2 cache when the block is replaced, invalidated, upgraded or downgraded from the L1 cache.

4.5. Possible Inaccuracies in Coherence/Noncoherence Miss Identification

The relatively simple mechanism to detect coherence misses described in Section 3.1 can err in two ways. First, on power-up or after a page fault, the state of the block is set to invalid, but the tag need not be initialized and may accidentally match a requested block. This should be quite rare and is random enough to not attribute

[8]This technique of detecting false sharing through data comparison was used by Coherence Decoupling [Huh et al. 2004] to save cache access latency due to false sharing misses.

[9]Note that, if AcT has no state for a block (e.g., in Partial AcT), we can omit allocation of OD state without further loss in accuracy.

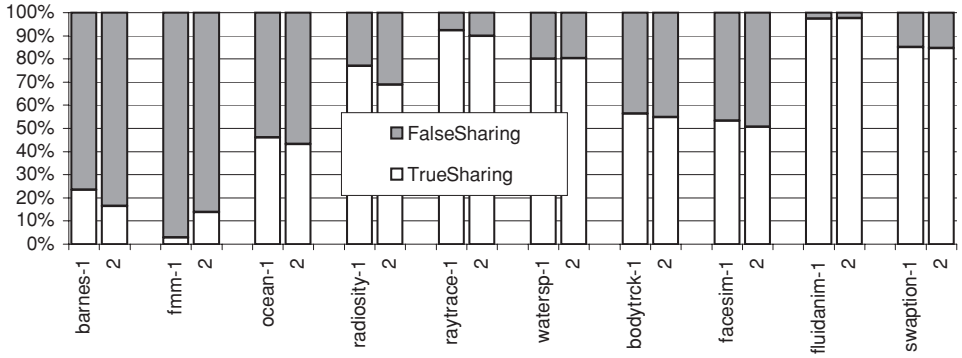


Fig. 9. Breakdown of coherence misses. For each benchmark, the two bars represent breakdown of coherence misses on caches that (1) always replace LRU block, and (2) prioritize invalid blocks over LRU block.

significant numbers of misses to any particular piece of code. Second, cache replacement 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 a block to be replaced between its invalidation and the subsequent miss, so replacement priority is not expected to obscure enough coherence misses to hide a scalability problem. Figure 9 validates this expectation quantitatively, by comparing cache miss classification results in caches that do and caches that do not prioritize replacement of invalid blocks. We find that prioritization of invalid blocks has only a small effect on the breakdown of true and false sharing misses and hence is unlikely to mislead programmers.

5. RELATED WORK

True and false sharing misses have been defined by Torrellas et al. [1990], Eggers and Jeremiassen [1991], and Dubois et al. [1993]. Each of them also describe an offline classification algorithm. Bianchini and Kontothanassis [1995] and Foglia [2001] have proposed algorithms for classification of coherence-related overheads for different protocols based on offline analysis of data access patterns. In contrast to these schemes and definitions, the mechanisms we describe in this article are designed to be implemented in real hardware for integration with existing online performance debugging infrastructures. Tools such as MemSpy [Martonosi et al. 1992], SIGMA [DeRose et al. 2002], SM-prof [Brorsson 1995], PIN [Luk et al. 2005], and Valgrind [Seward 2004] are software-only performance debuggers to study memory bottlenecks. Unlike these simulation-based tools, which suffer major performance overheads, our coherence miss classification mechanisms are intended to classify cache misses at-speed and drive performance counters for performance debugging of real applications with real input sets running on actual hardware.

Coherence Decoupling [Huh et al. 2004] uses local data comparison that we describe in Section 4.2 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. In our design space, coherence decoupling maps to Zero OD with Zero AcT.

Prior works have also looked at avoiding false sharing misses by associating coherence protocol information with cache subblock units [Dubnicki and LeBlanc 1992; Goodman 1987]. This type of design, also known as sectored cache, needs to

Table II. Splash-2 Benchmarks and Their Inputs

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

Table III. PARSEC Benchmarks and 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

accommodate coherence information for subblock units at various granularities for a given cache block size. Also, the coherence protocol implementation should be modified extensively on top of changes to the existing cache hardware. On the contrary, our scheme is aimed at providing feedback directly to the programmers through minimal hardware changes that are off the critical path that could affect program performance.

Numerous research proposals have been made for improving the performance counter infrastructure [Sastry et al. 2001], attribution of performance-related events to particular instructions [Dean et al. 1997], and for sampling and processing of profiling data [Alexandrov et al. 2007; Mousa and Krintz 2005; Nagpurkar et al. 2006; Zhao et al. 2007; Zilles and Sohi 2001]. Our cache miss classification mechanisms are synergistic with improvements in performance counters, sampling, and profiling infrastructure: better profiling infrastructure increases the value of our classification to the programmer, and our scheme enhances the value of a profiling infrastructure by providing additional event types that can be profiled.

Much work has also been done in sofware tools for multithreaded correctness and in thread-safety of tools themselves. Chung et al. [2008] proposed a dynamic binary translation framework and use transactional memory mechanisms to detect and correct data races in multithreaded programs. Nagarajan and Gupta [2009] propose architectural support for exposing cache-related events to software; this enables better control of interprocessor shared memory dependences for the programmer as well as handle misspeculation when speculatively executing past barriers. Ruwase et al. [2010] explore compiler-based optimizations to reduce runtime overheads for dynamic correctness checking mechanisms.

6. EVALUATION SETUP

We evaluate our DeFT mechanisms using SESC [Renau et al. 2006], a cycle-accurate execution-driven simulator, to model a 64-core chip-multiprocessor with 2.93GHz 4-way out-of-order cores, each with a private 32KB, 4-way set-associative L1 cache and a private 256KB, 16-way set-associative L2 cache. L2 caches are kept coherent using the MESI protocol, and all cores share an 8MB, 32-way L3 cache. The block size is 64 bytes in all caches.

For partial state schemes that hold a limited number of entries, a replacement policy is needed to make room for incoming blocks. We use the NkMRU(Not “k” Most Recently Used) replacement policy, which replaces a random entry that is not among the “k” most-recently-used cache blocks. It should be noted that not all invalidated or downgraded cache blocks will eventually suffer coherence misses, so we promote an

entry into the k most-recently-used set only when its cache block actually suffers a coherence miss in one of the caches.

We use two sets of benchmarks for our evaluation: Splash-2 benchmarks [Woo et al. 1995] (Table II) and all benchmarks from the PARSEC-1.0 [Bienia et al. 2008] benchmark suite that we could compile to run in our simulator (Table III) Both benchmark suites are highly scalable and thoroughly tuned. Therefore, our experiments show the accuracy of our DeFT mechanisms with respect to I-FSD in these applications, but we do not expect to find actual scalability problems. We measure accuracy (how many misses are misclassified) and perform attribution to individual program counters (static instructions) to see how closely attribution results for each DeFT scheme match those reported by I-FSD. This helps us to determine whether the programmer would still get a meaningful picture about false and true sharing in the program.

We estimate chip area overheads of DeFT mechanisms schemes using Cacti 5.3 [Jouppi et al. 2006], an integrated cache access time, area and dynamic power model. These area costs come from per-core OD and AcT state maintained by L1 and L2 caches as well as global AcT state in the L3 cache.

Our DeFT schemes could result in performance overheads in two ways: (1) when primary caches update per-core read/write vectors for every access, there is a 0.41% latency increase for 32 KB L1 caches used in our experiments and (2) when the on-die interconnect becomes saturated with extra traffic from AcT state needed for reading and updating information in the central repository or global pool. All other latencies for AcT and OD state updates can be hidden by looking up state in parallel with data accesses and/or updating state after data access completes. We did not find any instances of interconnect saturation, and a $<$1% L1 latency increase is unlikely to affect performance. Because our mechanisms are not expected to affect performance, we do not show any performance overhead results.

We do not have access to an existing software tool that implements I-FSD or another false sharing detection algorithm to directly collect accuracy or performance data, and building such a tool is beyond the scope of this article. We therefore concentrate on DeFT design points in our evaluation. However, we try to give the readers a sense for what we could expect from a software tool. Regarding accuracy, we are not aware of any cache miss classification algorithm whose accuracy (in the programmer-centric sense) matches or exceeds I-FSD. Regarding performance, we cite performance overheads of other sofware-based memory analysis tools; we expect a software false sharing detector tool to have similar overheads.

7. EXPERIMENTAL EVALUATION

In this section, we first show that coherence misses become more prevalent as we increase the number of cores. We then quantitatively examine how classification accuracy and hardware cost of DeFT schemes is affected by approximations along the AcT and OD axes for 64 cores. Based on this analysis, we also examine specific AcT-OD combinations whose cost is low enough for actual hardware implementation.

When evaluating the accuracy of DeFT schemes, we use our I-FSD algorithm as the baseline, and measure two aspects of potential inaccuracy. First, we measure on how many misses there is disagreement (true or false sharing miss) between the particular DeFT scheme and I-FSD. Second, some of the DeFT schemes (Partial AcT or OD state) cannot classify all coherence misses, so we measure the percentage of coherence misses that are actually classified. We note that this inability to classify all coherence misses effectively creates a sampling method, and results in inaccurate reporting to the programmer only when the sample (coherence misses that are classified) is not sufficiently representative of the population (all coherence misses). Also, all coherence misses are

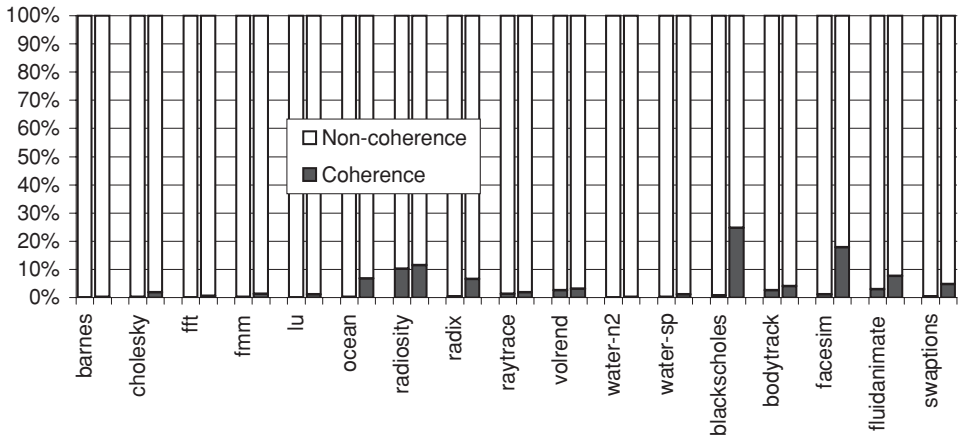


Fig. 10. Breakdown of all cache misses for 8 and 64 cores.

detected by DeFT schemes, but classification of these coherence misses uses sampling in schemes that have partial state.

7.1. Scaling of Coherence Misses

Figure 10 shows the breakdown of cache misses into coherence and noncoherence misses as we increase the number of cores from 8 to 64 for our benchmarks. With more cores, coherence misses represent a larger percentage of all cache misses. As a result, performance issues related to true and false sharing become worse when more cores are used, and are more likely to be a scaling limiter. As a result, tools and mechanisms for identification and classification of coherence misses are likely to be increasingly relevant in the near future.

7.2. Varying AcT state

Figure 11 shows accuracy and classification rate results for five different AcT mechanisms, using Total OD in combination with each AcT mechanism. This figure shows, for each benchmark (from left to right), Near-Total AcT (global AcT state kept for every block in the shared L3 cache, with a total of 256k entries), three Partial AcT design points with global AcT pools of 16k, 4k, and 1k entries, and Zero AcT (AcT information inferred using data comparison).

As expected, Near-Total AcT achieves nearly perfect accuracy. The few disagreements ($<$1% of coherence misses in ocean, none in other applications) are caused by situations similar to the one shown in Figure 5. Partial AcT with 16k and 4k entries omit classification of relatively few ($<$10% except for fmm) coherence misses and achieve excellent ($>$90%) agreement with I-FSD, so we expect low distortion of results reported to the programmer. Partial AcT with 1k entries result in some reduction in actual accuracy and fewer classifications (more potential for sampling error). Finally, Zero AcT has no sampling error but has substantial disagreement with I-FSD, resulting in incorrect classification of 4.5% (in radix) to 33.5% (in barnes) of coherence misses. To estimate the actual impact on the programmer of both inaccuracy (disagreement with I-FSD) and sampling (inability to classify), we perform an experiment where we assume that the programmer should be provided with a breakdown of all coherence misses by static instruction and, for each static instruction, by type (true or false sharing). This is consistent with the most common usage of performance counters today, and will tell the

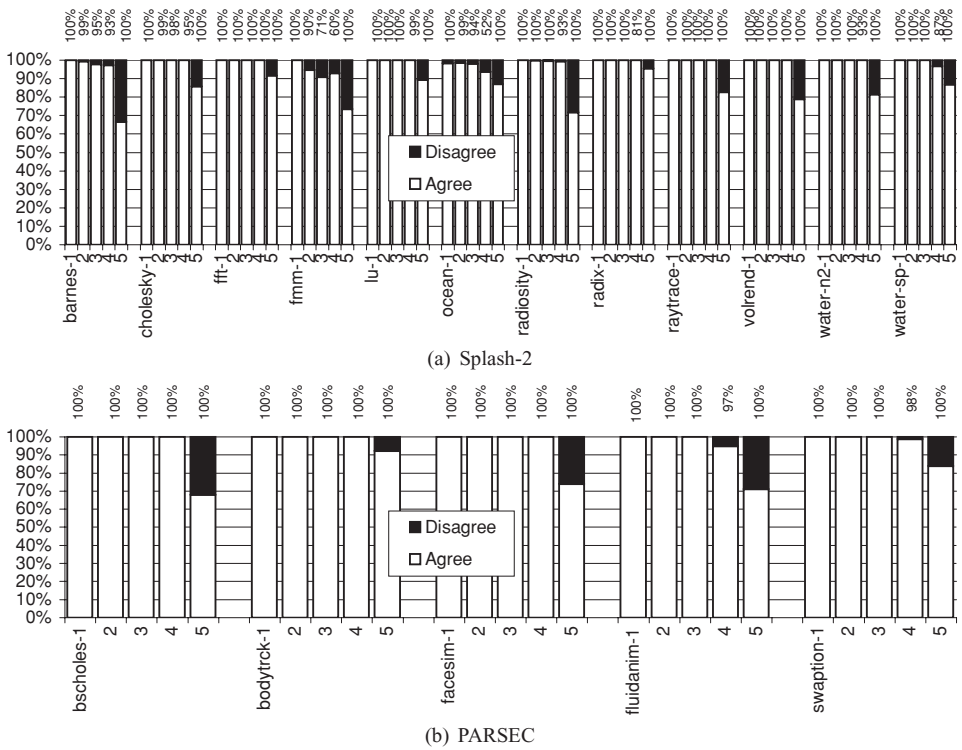


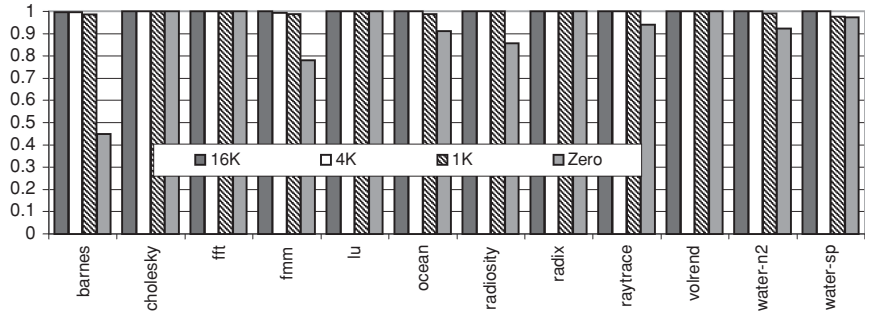
Fig. 11. Accuracy for different AcT approximations with classification rates (% of coherence misses that are actually classified) shown above each bar. For each benchmark, the five bars (from left to right labeled 1 through 5) represent Near-Total AcT state, Partial AcT state with 16k, 4k, 1k entries and Zero AcT state.

programmer both: (1) which instructions (lines of code) are causing most of the misses and (2) the dominant types of misses for each instruction.

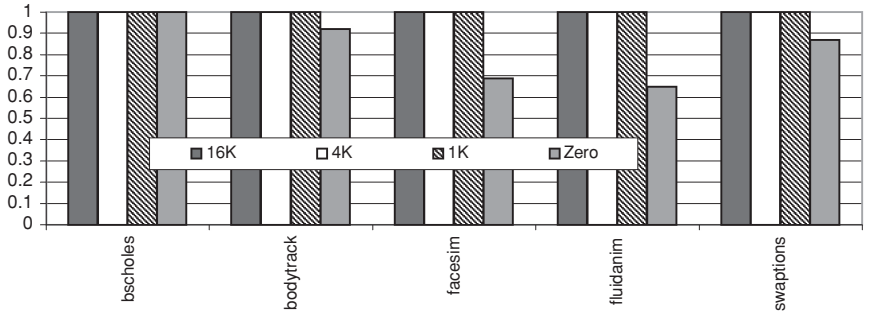
Figure 12 shows, for each AcT scheme, the Pearson's correlation coefficient between the scheme's report and the report from I-FSD. We observe that even 1k-entry Partial AcT schemes, despite having a significant fraction of coherence misses go unclassified, still provide reports that nearly perfectly correlate with I-FSD reports for all applications.

A final observation from Figure 12 is that Zero AcT often produces reports that do not have very high correlation with correct (I-FSD) reports. We examined actual Zero AcT reports and found that in several applications the top few (e.g., ten) offenders in each category (false and true sharing) in Zero AcT and I-FSD still mostly match for many (but not all) applications, but in most applications there are significant differences in percentages in each category for a particular instruction. Overall, Zero AcT would often still lead to correct conclusions (which kind of fixes are needed and where they should be made), but in some applications the results would likely mislead programmers into attempting (time consuming) fixes that do not improve (and may even hurt) performance.

Figure 13 shows the area cost of global AcT state for each AcT mechanism, shown as a percentage of total on-die cache area. Note that the total cost of a DeFT scheme is a combination of global AcT cost (shown here) and per-core AcT and OD cost which will be discussed in Section 7.3. The most accurate Near-Total AcT scheme keeps global AcT state for all words in the shared L3 cache, which represents a 4.36% overhead.



(a) Splash-2



(b) PARSEC

Fig. 12. Correlation coefficient between number of false sharing misses suffered by static instructions in Partial AcT, Zero AcT schemes against Near-total AcT. Samples of static instructions are chosen by Partial AcT schemes.

Near-Total	Partial (16k)	Partial (4k)	Partial (1k)	Zero
4.22%	1.68%	1.59%	1.25%	0.00%

Fig. 13. Area overhead of AcT state as a percentage of *on-die cache area*. Total OD state adds 7.35% area overhead to on-die caches.

Partial AcT reduces AcT cost to about 1.7%, 1.6%, and 1.25 for 16k, 4k, and 1k-entry global AcT pools, respectively. Finally, the Zero AcT scheme has no global AcT state (cost of global AcT support is zero).

Since Zero AcT is the most desirable mechanism in terms of cost, we conducted additional experiments to gain insight into what is causing its inaccuracy. We modified Total AcT to ignore updates to *Reader* bits in global AcT state (to model Zero AcT's lack of such state) and modified Total AcT to not update the Writer field in global AcT state for silent writes (to model Zero AcT's inability to identify such writes). We find that nearly all of Zero AcT's inaccuracy is due to these two limitations, but that neither of the two is dominant in all applications: missing read information is the main culprit in barnes, fmm, radiosity, and blacksholes, both limitations are about equally responsible in volrend and facesim, and missing silent writes is the dominant problem in the remaining benchmarks.

7.3. Varying OD state

Figure 14 shows accuracy and classification rates for four different OD mechanisms, using Near-Total AcT state. The figure shows, for each benchmark (from left to right): Total OD (OD state kept for every cache block with a total of 4k entries for each private

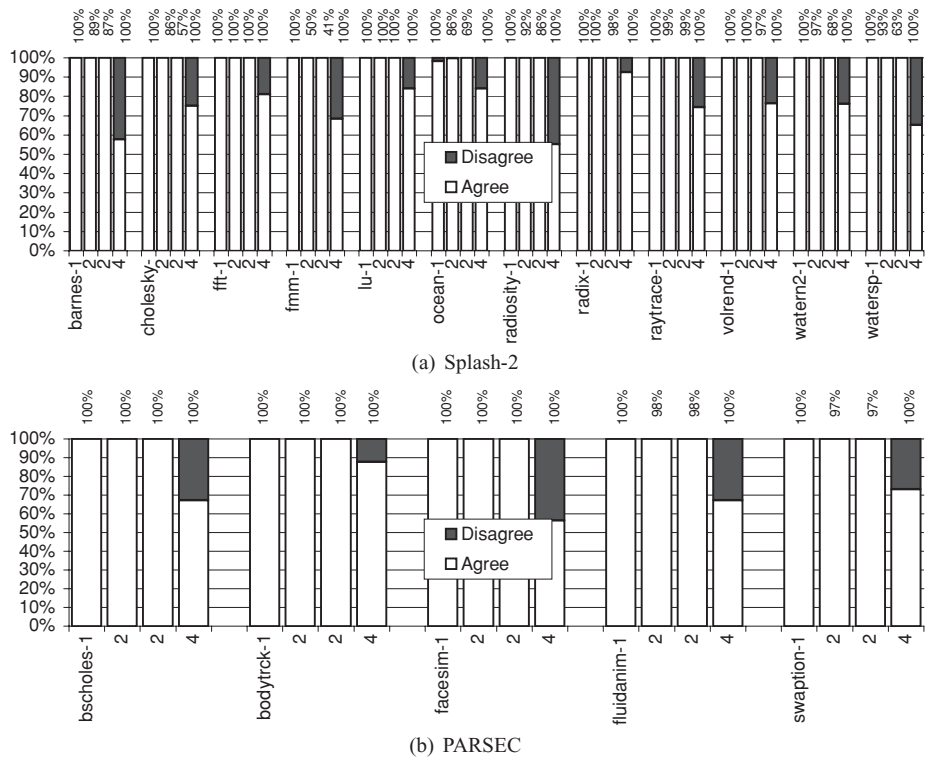


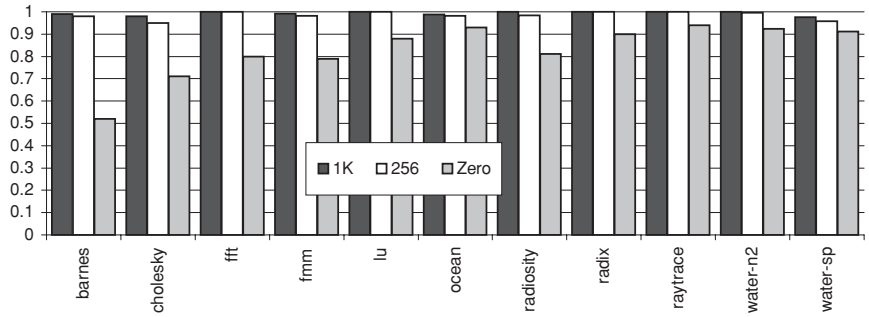
Fig. 14. Accuracy for different OD approaches with classification rates (% of coherence misses that are actually classified) shown above each bar. For each benchmark, the four bars (left to right labeled 1 through 4) represent Total OD state, Partial OD state with 1k and 256 entries, and Zero OD state.

cache), two Partial OD design points with per-core pool of 1k, and 256 entries, and Zero OD (false and true sharing are detected at the time of a coherence miss).

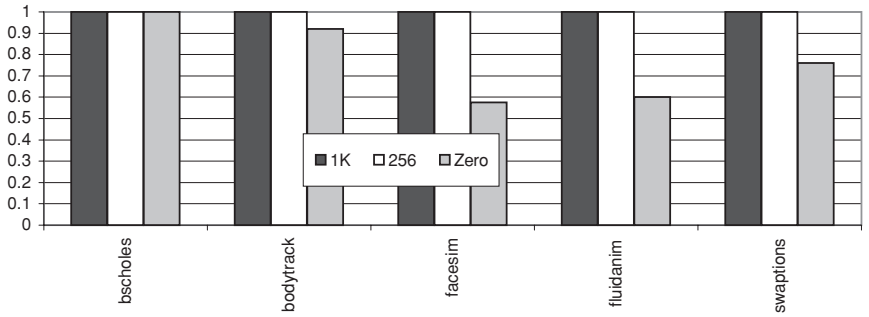
Total OD state achieves the highest accuracy. Partial OD schemes achieve nearly perfect accuracy (>99% agreement with I-FSD). However, Partial OD schemes provide fewer classifications in some cases (up to 60%, for fmm) and hence may have sampling error. Zero OD has no sampling error but has substantial disagreement with I-FSD, resulting in incorrect classification of 15% (bodytrack) to 80% (radiosity) coherence misses. This disagreement between Zero OD and I-FSD is caused by missing true sharing accesses that occur after the time of coherence miss.

Figure 15 shows correlation between reports produced by different OD schemes and the report produced by I-FSD. We observe that Partial OD schemes, despite having a significant fraction of coherence misses go unclassified in some applications, still provide reports that nearly perfectly correlate with I-FSD reports for most applications. However, Zero OD often produces reports that do not have high correlation with correct (I-FSD) reports. We examined actual Zero OD reports and had similar findings as we did for Zero AcT; the results would likely lead to similar conclusions in most applications, but in several applications they are likely to mislead the programmer.

Figure 16 shows the area cost of OD state for each DeFT mechanism, shown as a percentage of total on-die cache area. Again, note that the total cost of a DeFT scheme is a combination of per-core OD cost (shown here) and the AcT cost (shown in Section 7.2). The Total OD scheme keeps OD state for all words in private caches, which represents a 7.35% overhead. Partial OD schemes reduce OD cost to about 3.8% for 1k and to 2%



(a) Splash-2



(b) PARSEC

Fig. 15. Correlation coefficient between number of false sharing misses suffered by static instructions in Partial OD, Zero OD schemes against Total OD. Samples of static instructions are chosen by Partial OD schemes.

Total	Partial(1k)	Partial(256)	Zero
7.35%	3.77%	1.99%	0.0%

Fig. 16. Area overhead of OD state as a percentage of *on-die cache area*. Total global AcT state adds 4.22% area overhead to on-die caches.

for 256-entry per-core pools. Finally, the Zero OD scheme has no OD state (cost of OD support is zero).

7.4. Specific Low-Cost AcT-OD Combinations and Dubois et al.'s Scheme

Figure 17 shows the accuracy and sampling rates for a range of DeFT schemes that have relatively low cost and can be implemented in real hardware. For each benchmark, we show four different schemes (from left to right): (1) Partial (16k) AcT with Partial (1k) OD, (2) Partial (4k) AcT with Partial (256) OD, (3) Zero AcT with Zero OD, and (4) Dubois et al.'s scheme.

From prior experiments, we have seen that Partial AcT and Partial OD on their own offer reasonable accuracy while achieving lower costs. Our experiments show that, when Partial AcT and Partial OD are combined, they still retain very good accuracy. However, we observe that the total inaccuracy of a combined (Partial AcT, Partial OD) scheme is slightly worse than the sum of their individual inaccuracies from Figures 11 and 14. This is because we couple per-core AcT state (Own-Access bits) with per-core OD state into the same entry, to avoid having separate structures (pools) for per-core Act and per-core OD state. This means that, when combining an AcT scheme with a Partial OD scheme, the accuracy of AcT may suffer additionally because some accesses are not captured (and thus do not eventually update global AcT state). How-

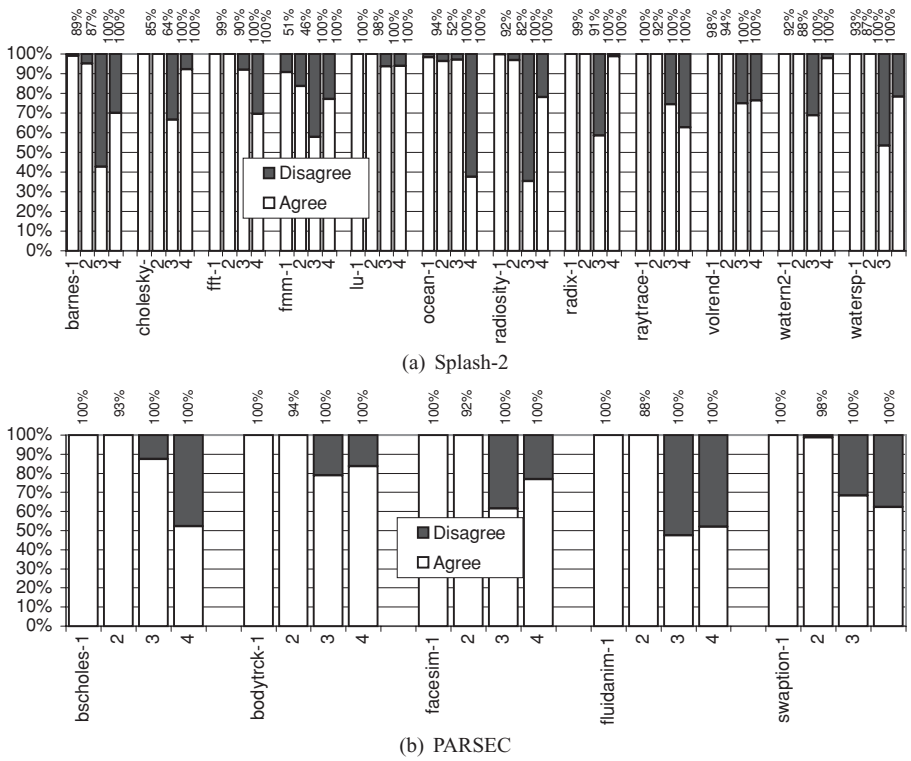
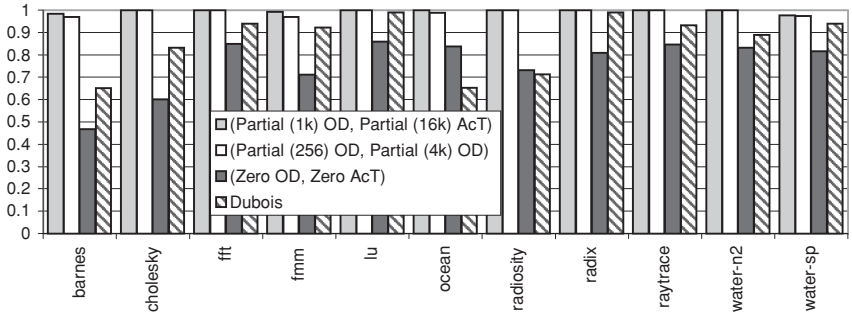


Fig. 17. Accuracy for low-cost DeFT and Dubois et al.'s schemes with classification rates (% of coherence misses that are actually classified) shown above each bar. For each benchmark, bar 1 (leftmost bar in each 4-bar group) represents (Partial (16k) AcT, Partial (1k) OD), bar 2 is (Partial (4k) AcT, Partial (256) OD), bar 3 is (Zero AcT, Zero OD), and bar 4 is Dubois et al.'s scheme.

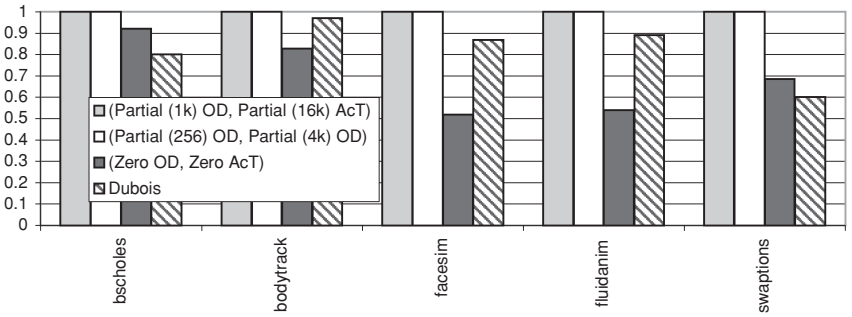
ever, even with this effect, the accuracy of (Partial AcT, Partial OD) is still very good, and Figure 18 shows that their reports still very strongly correlate with I-FSD reports.

The (Zero OD, Zero AcT) design point, which is attractive because of its nearly-zero hardware cost, is the least accurate. It misclassifies up to 65% (for radiosity) of coherence misses, and its reports have poor correlation with I-FSD reports in many applications. We note that this design point was used in Coherence Decoupling [Huh et al. 2004] to perform value speculation and hide cache miss latency due to false sharing. Although this design point was successfully used for speculation (where there are recovery mechanisms), it is unlikely to be suitable for performance debugging because it often produces reports that are likely to mislead the programmer.

We also show the Dubois et al.'s [1993] scheme for comparison. Note that the accuracy of this scheme is also shown with respect to I-FSD, and that most of the disagreement is caused simply by differences in how false sharing misses are defined in I-FSD and in Dubois et al. However, one conclusion that can be drawn from this data is that, for programmer-centric purposes, the Dubois et al.'s scheme only achieves accuracy similar to our nearly-cost-free (Zero OD, Zero AcT) DeFT scheme. Among the two, (Zero OD, Zero AcT) has better accuracy in benchmarks with large numbers of false sharing misses on the writer side, whereas Dubois et al. achieves better accuracy when silent and idempotent writes (which are not detected as writes in the Zero OD approximation) cause true sharing misses.



(a) Splash-2



(b) PARSEC

Fig. 18. Correlation coefficient between number of false sharing misses suffered by static instructions in various schemes against (Total OD, Near-Total AcT). Samples of static instructions are chosen by partial state schemes.

Partial(1k) OD Partial(16k) AcT	Partial(256) OD Partial(4k) AcT	Zero OD Zero AcT	Dubois et al. scheme
5.51%	3.07%	0.0%	3.06%

Fig. 19. Area overhead of partial state DeFT and Dubois et al. schemes as a percentage of *on-die cache area*.

Finally, Figure 19 shows the area cost incurred by DeFT schemes, as a percentage of total on-die cache area. Partial schemes incur relatively modest area overheads of about 5.5% and 3%, a major portion (about 70%) of which is incurred by maintaining per-core pools for every private cache in our 64-core configuration. The (Zero OD, Zero AcT) scheme can be implemented with minimal hardware modifications, such as changing the cache controller to perform data comparison on a coherence miss. For Dubois et al.'s scheme, a stale bit is needed for every word in every cache to track whether the fresh value produced by a write has been consumed. For every 32-bit data word in caches, a 1-bit overhead is incurred. Note that (Partial(256) OD, Partial(4k) AcT) has similar cost to Dubois et al. but achieves consistently better programmer-centric accuracy. Overall, we conclude that Partial AcT - Partial OD schemes can offer very good accuracy with reasonably modest costs, while the Zero AcT - Zero OD scheme can offer some insight that can benefit experienced programmers with nearly-zero hardware cost.

8. CONCLUSIONS

Achieving good parallel performance is critical to the success of current and future multi- and many-core processors, but it is currently very difficult for programmers to diagnose (and therefore fix) a common scalability problem: excessive coherence misses. Programmers must not only find the source (line of code) of cache misses but, in order

to address them, must also discover whether the misses are due to true sharing, false sharing, or noncoherence (usually capacity) reasons.

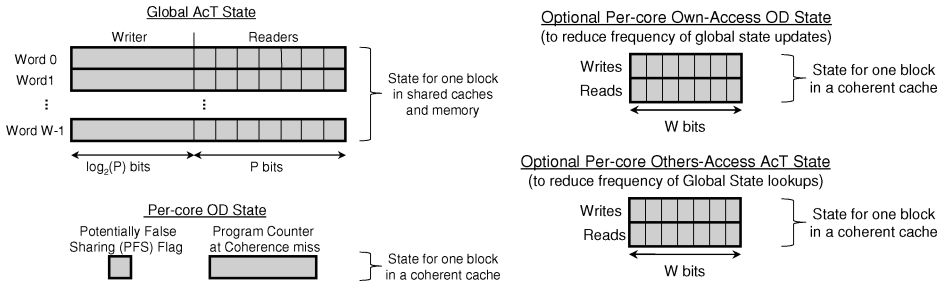
In this article, we introduce a programmer-centric definition of true and false sharing, an algorithm to classify coherence misses according to this definition, and approximations that makes this definition suitable for implementation in real processors, to drive hardware performance counters. We evaluate a range of design points to explore the trade-offs between hardware cost, classification accuracy, and impact on the programmer. We find that it is possible to achieve very good accuracy with relatively low cost (about 3% of total on-chip cache area), which is only a fraction of the cost needed for the ideal (most expensive) scheme. We also find that the simplest scheme, which keeps no state at all, can be implemented at nearly zero cost, yields good results in some benchmarks, but is likely to mislead the programmer in others.

APPENDIX

A. IDEAL FALSE SHARING DETECTOR (I-FSD) ALGORITHM

In this section, we present a pseudocode algorithm based on the programmer-centric definition for false sharing coherence misses described in Section 3.3.

```
//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
```




```
PROC ACCESS_TRACKING
 //Tracks reads and writes by all cores through Global and Local vectors

 ON Coherence Miss to B
   WAIT ON current sharers to update Global AcT State;

 ON Write access to A
    Own_Access_vector_write[A] = 1;
    Own_Access_vector_read[A] = 0;

 ON Read access to A
    Own_Access_vector_read[A] = 1;

 ON Invalidation/Downgrade/Replacement/Upgrade request for B
    FOR each word address K in B DO
     IF (Own_Access_vector_write[K]==1) THEN
       Global_Writer_ID = c;
       Global_Reader_vector={0};
     ENDIF
     IF (Own_Access_vector_read[K]==1) THEN
       Global_Reader_vector[c]=1;
     ENDIF
    DONE

ENDPROC ACCESS_TRACKING
```

```
PROC OVERLAP_DETECTION
 //Classifies Coherences misses through Others vector, Program Counter,
 //Coherence_Miss and Potential_False_Sharing

 ON Coherence Miss to B
   Set Coherence_Miss to true;
   Set Potential_False_Sharing to true;
   //Reset on observing first true sharing on the block
   Record the current PC into Miss_PC;
   Others_Access_vector_write = Others_Access_vector_read = {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
     Others_Access_vector_write[K] = 1;
    ENDIF
    IF (a bit other than c is set in Global_Reader_vector) THEN
     Others_Access_vector_read[K] = 1;
    ENDIF
   DONE

 ON Write access to A
   IF (Coherence_Miss == true AND
       Potential_False_Sharing == true) THEN
    //True sharing if another core had read(consumer) or
    //written(producer) to this word
    IF (Others_Access_vector_write[A] == 1 OR
        Others_Access_vector_read[A] == 1) THEN
     Potential_False_Sharing = false;
    ENDIF
   ENDIF

 ON Read access to A
   IF (Coherence_Miss == true AND
       Potential_False_Sharing == true) THEN
    //True sharing if another core had written(producer) to this word
    IF (Others_Access_vector_write[A] == 1) THEN
     Potential_False_Sharing = false;
    ENDIF
   ENDIF

 ON Invalidation/Downgrade/Replacement/Upgrade request for B
   IF (Coherence_Miss == true) THEN
    Record Miss_PC and Potential_False_Sharing; //Output to Profiler
   ENDIF

ENDPROC OVERLAP_DETECTION
```

REFERENCES

ALEXANDROV, A., BRATANOV, S., FEDOROVA, J., LEVINTHAL, D., LOPATIN, I., AND RYABTSEV, D. 2007. Parallelization made easier with Intel performance-tuning utility. *Intel Technol. J. 11*, 4.

BIANCHINI, R. AND KONTOTHANASSIS, L. 1995. Algorithms for categorizing multiprocessor communication under invalidate and update-based coherence protocols. In Tech. rep. 533, University of Rochester.

BIENIA, C., KUMAR, S., SINGH, J., AND LI, K. 2008. The PARSEC benchmark suite: Characterization and architectural implications. Tech. rep. TR-811-08, Princeton University.

BRORSSON, M. 1995. Sm-prof: A tool to visualise and find cache coherence performance bottlenecks in multiprocessor programs. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'95/PERFORMANCE'95)*. ACM, New York, 178–187.

CHUNG, J., DALTON, M., KANNAN, H., AND KOZYRAKIS, C. 2008. Thread-Safe dynamic binary translation using transactional memory. In *Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture*. 279–289.

DEAN, J., HICKS, J. E., WALDSPURGER, C. A., WEIHL, W. E., AND CHRYSOS, G. Z. 1997. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the International Symposium on Microarchitecture*. 292–302.

DEROSE, L., EKANADHAM, K., HOLLINGSWORTH, J. K., AND SBARAGLIA, S. 2002. Sigma: A simulator infrastructure to guide memory analysis. In *Proceedings of the ACM/IEEE Conference on Supercomputing (Supercomputing'02)*. IEEE Computer Society Press, Los Alamitos, CA, 1–13.

DUBNICKI, C. AND LEBLANC, T. J. 1992. Adjustable block size coherent caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*. IEEE Computer Society Press, Los Alamitos, CA, 170–180.

DUBOIS, M., SKEPPSTEDT, J., RICCIULLI, L., RAMAMURTHY, K., AND STENSTROM, P. 1993. The detection and elimination of useless misses in multiprocessors. Tech. rep. CENG 93-02, USC.

EGGERS, S. AND JEREMIASSEN, T. 1991. Eliminating false sharing. In *Proceedings of the International Conference on Parallel Processing*. 377–381.

FOGLIA, P. 2001. An algorithm for the classification of coherence related overhead in shared-bus shared-memory multiprocessors. *IEEE TCCA Newsl.*. IEEE Computer Society Press, Los Alamitos, CA.

GOODMAN, J. R. 1987. Coherency for multiprocessor virtual address caches. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'87)*. ACM, New York, 72–81.

HUH, J., CHANG, J., BURGER, D., AND SOHI, G. S. 2004. Coherence decoupling: Making use of incoherence. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 97–106.

INTEL. 2007. Debugging and performance monitoring. In *Intel64 and IA-32 Architectures Software Developer s Manual* Volume 3B: System Programming Guide, Part 2. Intel.

JOUPPI, N. P. ET AL. 2006. Cacti 4.2. http://quid.hpl.hp.com:9081/cacti/.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, 190–200.

MARTONOSI, M., GUPTA, A., AND ANDERSON, T. 1992. Memspy: Analyzing memory system bottlenecks in programs. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'92/PERFORMANCE'92)*. ACM, New York, 1–12.

MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J. 9,* 2, 78–117.

MOUSA, H. AND KRINTZ, C. 2005. Hps: Hybrid profiling support. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. IEEE Computer Society, Los Alamitos, CA, 38–50.

NAGARAJAN, V. AND GUPTA, R. 2009. Ecmon: Exposing cache events for monitoring. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, 349–360.

NAGPURKAR, P., MOUSA, H., KRINTZ, C., AND SHERWOOD, T. 2006. Efficient remote profiling for resource-constrained devices. *ACM Trans. Archit. Code Optim. 3,* 1, 35–66.

RENAU, J. ET AL. 2006. SESC. http://sesc.sourceforge.net.

RUWASE, O., CHEN, S., GIBBONS, P. B., AND MOWRY, T. C. 2010. Decoupled lifeguards: Enabling path optimizations for dynamic correctness checking tools. *SIGPLAN Not. 45,* 6, 25–35.

SASTRY, S. S., BODÍK, R., AND SMITH, J. E. 2001. Rapid profiling via stratified sampling. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. ACM Press, New York, 278–89.

SEWARD, J. 2004. Valgrind, An open-source memory debugger for x86-GNU/Linux. *http://valgrind.kde.org/*.

TORRELLAS, J., LAM, M. J., AND HENNESSY, J. L. 1990. Shared data placement optimizations to reduce multiprocessor cache misses. In *Proceedings of the International Conference on Parallel Processing*. 266–270.

WOO, S., OHARA, M., TORRIE, E., SINGH, J., AND GUPTA, A. 1995. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture*.

ZHAO, L., IYER, R., ILLIKKAL, R., MOSES, J., MAKINENI, S., AND NEWELL, D. 2007. Cachescouts: Fine-Grain monitoring of shared caches in cmp platforms. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*. IEEE Computer Society, Los Alumitos, CA, 339–352.

ZILLES, C. B. AND SOHI, G. S. 2001. A programmable co-processor for profiling. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA'01)*. IEEE Computer Society, Los Alamitos, CA, 241.

Received March 2010; revised September 2010; accepted November 2010