Dynamic Cache Contention Detection in Multi-threaded Applications

Qin Zhao David Koh Syed Raza
Saman Amarasinghe
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA, USA
{qin_zhao, dkoh, raza, saman}@csail.mit.edu

Derek Bruening
Google Inc
Mountain View
CA, USA
bruening@google.com

Weng-Fai Wong
School of Computing
National University of Singapore
Singapore
wongwf@comp.nus.edu.sg

Abstract

In today's multi-core systems, cache contention due to *true* and *false sharing* can cause unexpected and significant performance degradation. A detailed understanding of a given multi-threaded application's behavior is required to precisely identify such performance bottlenecks. Traditionally, however, such diagnostic information can only be obtained after lengthy simulation of the memory hierarchy.

In this paper, we present a novel approach that efficiently analyzes interactions between threads to determine thread correlation and detect true and false sharing. It is based on the following key insight: although the slowdown caused by cache contention depends on factors including the thread-to-core binding and parameters of the memory hierarchy, the amount of data sharing is primarily a function of the cache line size and application behavior. Using memory shadowing and dynamic instrumentation, we implemented a tool that obtains detailed sharing information between threads without simulating the full complexity of the memory hierarchy. The runtime overhead of our approach – a 5× slowdown on average relative to native execution – is significantly less than that of detailed cache simulation. The information collected allows programmers to identify the degree of cache contention in an application, the correlation among its threads, and the sources of significant false sharing. Using our approach, we were able to improve the performance of some applications up to a factor of 12×. For other contention-intensive applications, we were able to shed light on the obstacles that prevent their performance from scaling to many cores.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Optimization, Run-time environments

General Terms Performance

Keywords False Sharing, Cache Contention, Shadow Memory, Dynamic Instrumentation

```
int64 global_sum;
int64 local_sum[MAX_NUM_THREADS];
parallel_sum(int myid, int start, int end) {
  for (int i = start, i < end; i++)
    local_sum[myid] += buf[i];
  lock();
  global_sum += local_sum[myid];
  unlock();
}
```

Figure 1. Example code performing a parallel sum of elements in a buffer.

1. Introduction

One of the major success stories in modern computer architecture is the development of a memory hierarchy: the use of multiple levels of caches has helped bridge the large performance gap between processors and memory. A substantial body of work has focused on measuring and understanding cold, capacity, and conflict cache misses to better optimize applications for a given memory hierarchy. However, in the current multi-core era, additional cache misses can occur due to *true sharing* and *false sharing* in multi-threaded applications.

Threads:	1	2				
		Distinct cores			Same core	Padded
Time (s):	5.52	22.28	33.96	40.02	5.82	2.91

Table 1. Execution times of the parallel sum code from Figure 1 under different thread and core configurations.

Figure 1 shows a code fragment that computes the sum of elements in a buffer. Each thread adds up the elements in its assigned portion of the buffer, maintains the result in a private `local_sum` entry, and finally updates the `global_sum` atomically using its `local_sum`. Table 1 shows the execution times of this application on an eight-core (two quad-cores) machine under different configurations. When the application uses two threads running on separate cores, it is significantly *slower* than when it uses only a single thread or two threads on the same core. Furthermore, when the two threads run on different cores, the application has three possible execution times depending on how the threads are bound to different pairs of cores. These times range from 4× *to* 7× *slower* than the single-thread time. These results are quite surprising because the application has good parallelism and its threads only need to communicate at the end of their execution.

VEE'11, March 9–11, 2011, Newport Beach, California, USA.

In this example, unexpected slowdowns are caused by false sharing, which occurs because two threads repeatedly update the same cache line holding their private `local_sum` entries. If we add padding between `local_sum` entries to place them in different cache lines, the performance of the application with two threads running on different cores is substantially improved, and the expected linear speed-up is achieved (final column in Table 1). In addition, the three distinct execution times when the two threads run on different cores reflect the varying performance penalty incurred by false sharing depending on which two cores compete for the same cache line: cache line exchanges cost less on cores with a shared L2 cache than on cores without one. Thus, if thread contention was unavoidable for some reason, we could still reduce slowdowns by scheduling threads that frequently access the same data on cores with lower communication cost.

Even in this contrived example, while the solution is simple, the real challenge lies in precisely identifying performance bottlenecks in the application. Usually, this requires an understanding of a multi-threaded application's behavior and its interactions with the underlying hardware. To the best of our knowledge, no existing tool provides accurate and detailed information to help programmers identify and solve performance problems caused by cache contention in an efficient manner. Previous work generally relies on full cache simulation with prohibitively high runtime overhead. For example, CMP$im [15] runs at only 4-10MIPS.

The major novelty of this paper is identifying the minimal information required for cache contention detection, viz., cache line ownership, and building a set of tools to track cache line ownership and other information via shadow memory to help solve an important class of performance problems related to cache contention. In this paper, we propose a novel approach to obtain detailed information about cache contention by tracking the ownership of each cache line via memory shadowing and dynamic instrumentation. The basic scheme for efficiently tracking cache line ownership allows us to bypass the full complexity of the memory hierarchy, do away with expensive cache simulation, and hence drastically reduce the cost of obtaining sharing information. Contention detection, thread correlation, and false sharing detection are three examples built on top of the basic scheme.

1.1 Memory shadowing

Memory shadowing is a powerful technique for tracking properties of an application's data. It has been successfully used by many dynamic program analysis tools that need to maintain meta-data about every byte of the application's memory, including tools for detecting race conditions [13, 23, 29, 30], tracking dynamic information flow propagation [8, 22, 25], detecting memory usage errors [28, 31], and many others [5, 19, 20, 40]. General frameworks like Umbra [38] are developed to help users build customized memory shadowing tools. In this paper, we propose a new application of memory shadowing. By using shadow memory and dynamic instrumentation to track ownership of application memory at the granularity of cache lines, we are able to efficiently determine an application's sharing profile. Using appropriate meta-data, we can collect several types of information about thread interactions, and help programmers to identify and solve cache contention problems.

1.2 Contributions

This paper's contributions include:

- We present a novel scheme that, to the best of our knowledge, is the first scheme that can efficiently account for cache misses and invalidations caused by cache contention without resorting to expensive full cache simulation.
- We propose a novel use of shadow memory to track ownership of cache lines in application memory for cache contention analysis.
- We introduce a set of analysis tools that can determine the cache contention in applications, discover thread correlation, and detect true and false sharing, with an average overhead of $5\times$ slowdown compared to native execution.
- We were able to identify significant false sharing in 4 out of 18 real benchmark applications, and were able to improve the performance of 2 applications up to a factor of $12\times$.
- We discovered thread communication patterns and thread correlation in real applications via cache contention analysis.
- We showed that two benchmarks benefited from scheduling threads based on the memory hierarchy and their thread correlation, achieving up to a $2.4\times$ performance improvement.

1.3 Paper Layout

The rest of the paper is organized as follows: Section 2 provides an overview of inter-thread cache contention and describes our detection scheme. Section 3 discusses the details of our implementation. Section 4 presents experimental results from using our approach, Section 5 discusses related work, and Section 6 concludes the paper.

2. Overview

In multi-core systems, each individual core typically has a private L1 data cache for its own use. These private caches are the source of cache contention. When threads running on different cores access the same data or data within the same cache line, multiple copies of the cache line are created in the private caches of those cores. A hardware cache coherency mechanism is required to guarantee consistency between these copies. If a thread attempts to update a cache line that has been replicated in other cores' private caches, the hardware must invalidate all the other copies before the core can proceed with the write operation. A cache invalidation is an expensive operation as it causes memory operations to stall and wastes memory bandwidth. Moreover, cache invalidations can cause cache misses later when other cores access the same cache line again. Cache invalidations and misses can occur when cores compete to access the same data (*true sharing*) or different data items that happen to reside in the same cache line (*false sharing*). If cache invalidations and cache misses occur frequently, the performance of the application can suffer severely. It is therefore important for developers to be aware of sharing-induced slowdowns in order to scale their applications.

Traditionally, developers have had to resort to full cache simulation to obtain detailed cache behavior when studying capacity and conflict misses. Cache behavior can also vary significantly depending on the configuration of the memory hierarchy. However, we observe that the true/false sharing is only related to the size of a cache line and the application's behavior, and is independent of the other memory hierarchy parameters. This insight allows us to efficiently simulate cache contention behavior without running detailed cache simulations. Our approach dynamically tracks changes in ownership of application memory at the granularity of individual cache lines. This allows us to identify instructions that are responsible for the bulk of the cache misses and cache invalidations, threads that communicate frequently with each other, and interleaved accesses between different cores that cause cache contention.

2.1 Basic Scheme

We assume that our target application is running on a machine that has enough computational cores that each thread can run on

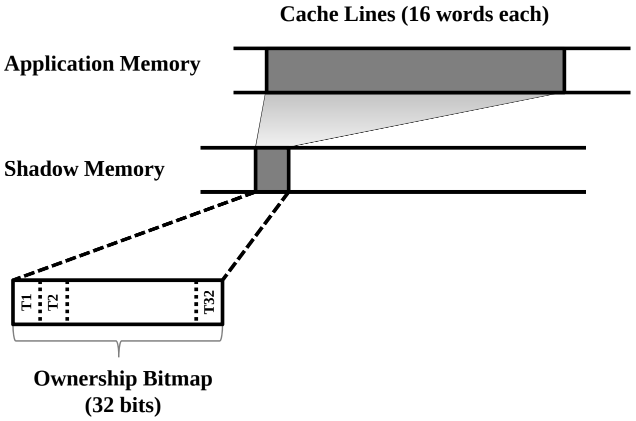

Figure 2. The shadow memory data structure used for our base contention detection scheme. Application memory is shadowed at cache line granularity with a bitmap indicating which cores contain a copy of that data.

its own dedicated core. This way, we can refer to threads and cores interchangeably, and report the *worst-case* results caused by contention. In this section we describe a design that assumes there are no more than 32 threads/cores simultaneously active in an application; Section 3 explains how our implementation scales to more threads.

We further assume that each core has a private L1 cache, and that data is never evicted from this private cache except through cache invalidations. This assumption allows us to avoid expensive cache simulation, ignore capacity and conflict misses, and isolate the inter-thread communication patterns that are sufficient for detecting cache contention. This assumption may also introduce inaccuracy due to not faithfully simulating the actual cache behavior, which will be discussed later in Section 3.5.

As shown in Figure 2, we use an *ownership bitmap* in shadow memory to track the distribution of each application memory cache line's copies. Each bit in the bitmap represents one core or thread. If it is set, it means that particular core has a copy of the associated cache line in its private cache, or that the core *owns* the cache line.

The bitmaps in the shadow memory are maintained by dynamically inserting instrumentation code for every application memory reference. When a thread reads a data object, it creates a copy of the cache line containing that data object in its own cache. We simulate this by setting the thread's bit in the cache line's ownership bitmap, indicating that the thread owns the cache line. When a thread updates an object, it invalidates all copies of the entire cache line in other cores' private caches, and updates the copy in its own cache. This is simulated by setting the current thread's bit and simultaneously clearing all other bits in the cache line's bitmap, thereby indicating the thread's exclusive ownership of this cache line.

Using this approach, we can simply examine ownership bitmaps at any time to observe the status of the cache lines. In addition to tracking cache line ownership, we often want to collect more information about an application in order to understand thread contention. In particular, we may wish to determine the degree of cache contention in the application, identify *delinquent instructions* that cause most of the cache contention events, determine the communication patterns and hence the correlation between threads, and compute the frequency of true/false sharing. In order to obtain such information, we need to add more instrumentation code and maintain additional information in shadow memory.

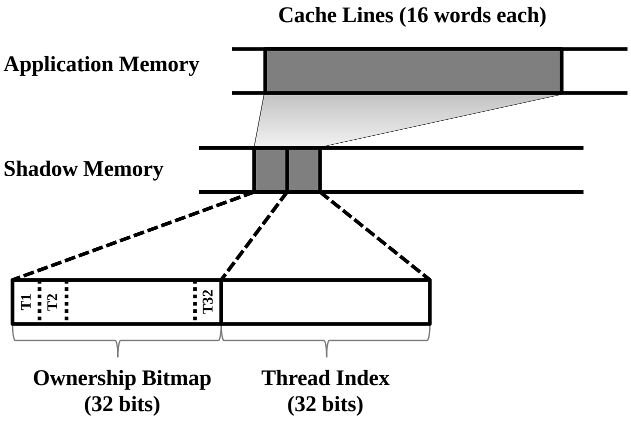

Figure 3. The shadow memory data structure used for determining thread correlation. A thread index is used to track which thread last updated the cache line.

2.2 Contention Detection

The first step in performance debugging is to check whether an application does indeed suffer from cache contention. Therefore, we may wish to determine the frequency of cache contention events (cache misses or cache invalidations) during program execution. To do so, we need to detect and count the total number of cache contention events, which can be easily done using the ownership bitmap of Figure 2.

A cache miss occurs when a core attempts to read a cache line that it does not own. In our scheme, this can be discovered by checking whether the current thread's bit is set in the corresponding ownership bitmap in shadow memory on a memory read. A cache invalidation happens when a core wants to update a cache line that it does not exclusively own. We can discover this by checking whether any bit other than the current thread's bit is set in the ownership bitmap. By checking the ownership of the target cache line on every memory reference, we can count the total number of cache contention events.

In addition, we can also track the behavior of individual memory access instructions. We associate each instruction with two thread-private counters that count the number of cache misses and invalidations it causes, respectively. By checking these counters after execution, we can easily identify the *delinquent access* instructions that are responsible for most of the cache misses and/or invalidations. Pinpointing such instructions is an important step in trying to reverse any performance losses due to cache contention.

2.3 Determining Thread Correlation

An effective optimization strategy is to identify groups of threads that communicate frequently with each other, and reduce the performance penalties of cache contention by scheduling them on cores with lower communication costs, e.g., cores with shared L2 caches.

We can extend our scheme to determine *thread correlation*. Two threads are *strongly correlated* if they communicate frequently with each other during the course of program execution. For instance, two threads may have a producer/consumer relationship and may need to communicate at fine granularity. To obtain thread correlation information, we allocate an array of counters for each thread to record its interaction frequency with other threads. We also add a field in the shadow memory unit to record the last thread that updated a data item, as shown in Figure 3. When a thread accesses a data item in a cache line, we identify which thread previously updated the cache line and increment the corresponding counter. We

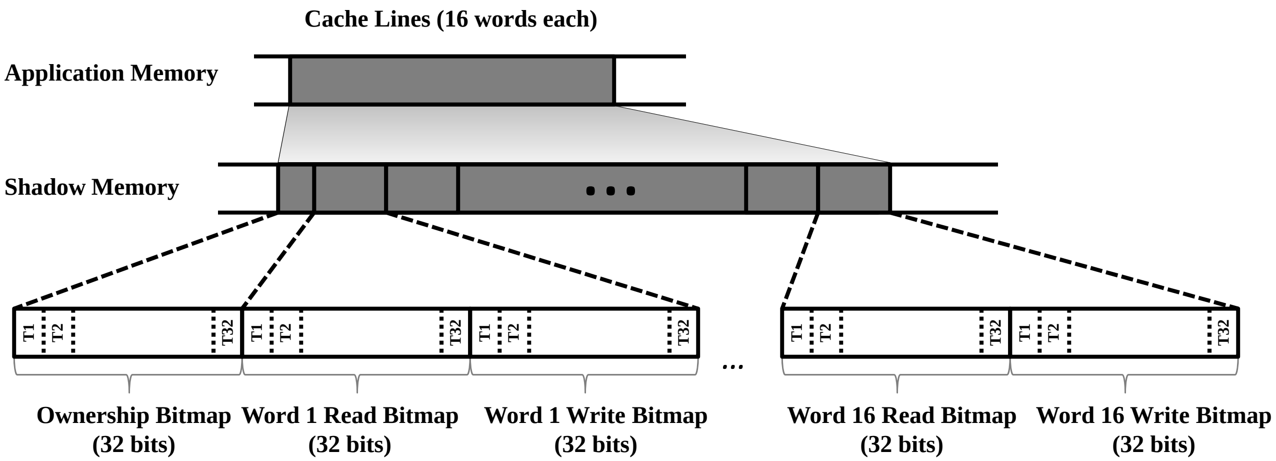
Cache Lines (16 words each)
Application Memory
Shadow Memory
T1
T2
T32
...
Ownership Bitmap (32 bits)
Word 1 Read Bitmap (32 bits)
Word 1 Write Bitmap (32 bits)
Word 16 Read Bitmap (32 bits)
Word 16 Write Bitmap (32 bits)

Figure 4. The shadow memory data structure used for false sharing detection. We track which threads read and wrote to each word in each cache line.

only update a counter when a cache miss or a cache invalidation occurs, so our correlation statistics only capture thread interactions that contribute to performance slowdowns. If there are two accesses to a piece of data and only one of the accesses causes a cache miss or invalidation, the counter is incremented only once. At the end of execution, using these counters, we can report the degree of correlation between threads, enabling the developer to act on the information and achieve better performance.

2.4 False Sharing Detection

Because cache lines are often larger than the data objects they contain, a single cache line often holds multiple data objects. Cache contention can arise due to true or false sharing of the cache line.

Definitions of false sharing are often imprecise or impractical for real-time detection [3, 18]. Here, we identify false sharing based on the following intuition: if a cache miss or cache invalidation could have been avoided had the target data been moved to another cache line, we consider the cache miss or invalidation to be caused by false sharing. More specifically, on a cache miss or invalidation, if the target location is also accessed by other threads, then it is a result of true sharing, and is unavoidable. Otherwise, it is result of false sharing.

In order to detect true/false sharing, we must maintain more information in the shadow memory for each cache line. As shown in Figure 4, we record the access history for each word within a cache line in the shadow memory. Specifically, we maintain a bitmap for each word within a cache line. Each thread has two bits in this bitmap to indicate whether it read and/or wrote the corresponding word. For every memory access, code is inserted to set the current thread's read or write bit in the bitmap for the specific word that was accessed.

During a delinquent access, we can determine if the cache miss or invalidation was caused by true or false sharing by examining the shadow memory of the corresponding cache line. On a cache miss, we can identify which threads updated this word by examining the bitmap associated with the word: the miss is due to false sharing if no other thread's write bit is set. Otherwise, it is due to true sharing. On a cache invalidation caused by a write, we check whether the target word was read or written by another thread by examining its bitmap once again. The invalidation is due to true sharing if some other thread's read or write bit is set. Otherwise, it is caused by false sharing. For a store, we clear all the bitmaps for the entire cache line in the shadow memory, and only mark the write bit of the current thread in the target word's bitmap. This is to capture the fact that the thread has exclusive ownership of the cache line, and to detect subsequent false or true sharing.

2.5 Further Extensions

Being architecture-independent, our approach can easily be applied to collect arbitrary information for a wide class of problems. Beyond the features described above, there are many other possible ways to extend the instrumentation for different purposes. For instance, we could add fields in the shadow memory to record which cache lines caused most of the cache invalidations, and report these problematic data locations to the user. It is also possible to add time stamps in the shadow memory to keep track of access times so that even more complex temporal relationships can be derived. Adding a field in shadow memory for storing call-stack context information is another possible extension to help programmers identify delinquent accesses with more context information.

Using a software-based approach lends our tool flexibility. First, it enables our tool to run on a wide range of commodity hardware. Second, our tool can change its parameters to analyze contention behavior over various thread and cache configurations that are different from the actual machine where the application is run. Finally, we can target different levels of the memory hierarchy; e.g., we can study not only private caches but also caches shared by a subset of cores by simply assigning the same thread bit to multiple threads.

3. Implementation

We implemented our detector on top of the shadow memory framework Umbra [38], which is an efficient memory shadowing framework that uses a novel mapping scheme [37] to translate application memory addresses to shadow memory addresses. Umbra is integrated with DynamoRIO [1, 4] and provides a simple API for client applications to easily add instrumentation to manipulate shadow memory without knowing the details of the underlying memory mapping scheme.

3.1 Base Framework

As described in Section 2, we associate each cache line (64 bytes) with a shadow memory unit in which a bitmap is used to represent which cores own the cache line. Using a 32-bit bitmap, we can track up to 32 threads or cores. We use Umbra's API to add instrumentation to update the shadow memory for every application memory reference. We instrument both reads and writes to keep track of cache line ownership changes in the bitmap. For each memory read, we set the bit for the current thread in the shadow memory bitmap corresponding to the cache line using an OR operation. For each memory write, we use a MOV instruction to set the bit corresponding to the current thread and clear all other bits at the same time.

Our implementation faced three main challenges. The first challenge is performance. The simple instrumentation described above could cause high runtime overhead. Because we must update the shadow memory on every memory access, a significant amount of bandwidth may be used by the shadow memory updates which will cause system bus congestion and stall execution. Moreover, the updates to the shadow memory itself will introduce high cache contention among threads. To avoid such problems, we optimized our instrumentation by adding checks and updating the shadow memory only when necessary (we call this approach *racy test and set*). In the common case, if multiple threads read but do not write the same cache line, only each thread's first read will cause a shadow memory update. In contrast, blindly writing the shadow memory (we call this approach *set*) even when no updates are necessary would cause different cores to compete on the cache line holding the shadow memory unit, incurring much higher runtime overhead. In addition, to minimize cache contention caused by accessing shadow memory, we ensure shadow memory units are cacheline-sized (64 bytes) or larger even though the actual meta-data in the shadow memory might be as small as 32 bits.

The second challenge is handling concurrent accesses to the bitmap. When multiple threads access the same cache line simultaneously, the corresponding instrumented code will simultaneously update the same bitmap. We could assign a lock to each bitmap and require any access to the bitmap to first acquire the lock (we call this approach *atomic test and set*). This guarantees that the bitmap check and update operations are atomic, but the runtime overhead is high. This is because the lock operation, an XCHG instruction in our implementation, is a very expensive operation that stalls the system in order to ensure atomic operation. In contrast, if we allow racy accesses, we may miss a few cache miss or cache invalidation events, but we will still catch most of them, which is sufficient for our analysis purposes. Our implementation uses racy accesses without locks (*racy test and set*).

The third challenge is scalability of design when more threads are added. Our current design uses one bit per thread in a 32-bit bitmap and is limited to 32 threads. One simple method is to recycle thread bits: e.g., represent thread 0 and thread 32 with the same bit. Bit recycling can lose precision and miscalculate cache misses or invalidations, because we cannot determine actual behavior when two threads represented by the same bit access the same cache line. Another method is to use a multi-word bitmap to track more threads. However, this approach incurs higher runtime overhead for its multi-word checks and updates. More importantly, using more words cannot scale up to a high number of threads. In real-world applications, some programs first create a group of threads to perform one task, destroy them, and then create another group of threads to perform the next task in sequence. Such programs can create hundreds of threads over the application lifetime, although only a few of them are simultaneously active. For such programs bit recycling works better than a multi-word bitmap scheme. It is possible to combine the two approaches to achieve both better precision and better scalability. In the prototype we implemented, we used bit recycling on a 32-bit bitmap.

3.2 Contention Detection

We extend the basic framework implementation described above to discover cache contention in an application. We associate two thread-private counters with each application memory reference instruction to count the cache misses and cache invalidations it causes in each thread, respectively. By doing so, we not only know the total number of cache misses and invalidations, but can also easily identify delinquent instructions, i.e., memory access instructions that cause most of the cache contention events.

For a memory read, we use a TEST instruction [1] to check whether the current thread has a copy of the referenced cache line (i.e., whether it owns the cache line). If the thread does indeed have a copy, no additional work is necessary. Otherwise, we can infer that there is a cache miss. Therefore, we update the cache miss counter and set the thread's bit in the shadow memory bitmap for that cache line using an OR instruction.

For a memory write, we use a CMP instruction to check if the only bit set in the bitmap for the cache line belongs to the current thread, i.e., whether it has exclusive ownership of the cache line. Again, if this is the case, no additional work is necessary. Otherwise, the inserted code will increment the cache invalidation counter and update the bitmap to reflect the thread's exclusive ownership with a MOV instruction.

When a cache invalidation occurs, we could determine whether the data has not been accessed by any other thread yet (a cold miss indicated by an all-0 bitmap), or whether there is at least one copy of the cache line in some other core (indicating a likely prior cache invalidation). However, this would increase the runtime overhead of our dynamic instrumentation. Because our focus is on inter-thread sharing contention rather than full cache simulation, each cache line can at most experience one cold miss in our model. Hence, we are trading off some accuracy for better performance. Similarly, for a memory read, the cache miss counter actually includes cache misses due to both cold misses and cache invalidations. It is possible to tell the difference between these two forms of misses by maintaining more information in the shadow memory. We make the same accuracy versus performance trade-off as before.

We also insert code to record the total number of instructions executed for each thread. When a thread terminates, we scan the two counters kept for each memory reference instruction to obtain the total number of cache misses and invalidations, and compare that against the total number of instructions executed by the thread to understand the extent of its cache contention. In addition, we can determine the delinquent instructions that contributed most to cache misses and cache invalidations for that thread. Where debug information is present, these instructions can be easily mapped back to the application's source code.

3.3 Determining Thread Correlation

To compute thread correlation, we allocate an array T_i of counters for each thread i. Counter $T_i[j]$ of the array corresponds to thread j. The first entry in the array is reserved for counting first-time accesses.

We also add a field in our shadow memory for each cache line that stores the index of T_i corresponding to the thread that performed the latest update to that cache line. On a cache miss or cache invalidation, we obtain the latest thread index from the shadow memory, and update the corresponding counter accordingly.

The stored counters can indicate a producer-consumer relationship. It is possible to separate cache misses from cache invalidations when updating these counters, so that producers can be distin-

[1] The TEST instruction performs an AND operation, updates the condition flags accordingly, and then discards the result.

guished from consumers. However, for our purposes, details about these relationships are not important, so we simply use one counter.

Because the memory is shadowed at the granularity of a cache line, our thread correlation actually reflects architectural-level correlation, which may or may not be equivalent to semantic correlation within the application. For example, if one thread updates one part of a cache line while another thread reads a different part of that cache line, our correlation counter is updated. This correlation is caused by false sharing, but it also affects performance, so reporting it is necessary.

In essence, our counters hold the frequency of a thread's interaction with every other thread. At the end of the application's execution, this correlation information is output to the user who may then use the information to understand the interaction among threads and schedule groups of correlated threads accordingly for better performance.

3.4 False Sharing Detection

In order to detect false sharing, we extend our shadow memory to record which threads have read and written to each word in a cache line.

Ideally, we should use 132 bytes of shadow memory for each 64-byte cache line to track 32 threads: 4 shadow bytes for recording cache line ownership, and 16 pairs of 4-byte bitmaps to track which of the 32 threads has read or written to each word in the cache line. However, Umbra does not currently support such a mapping scheme. Because we perform our experiments on an 8-core system, our prototype uses a compacted shadow memory data structure to track 8 threads using the configuration described below.

For each 64-byte cache line of the application, we maintain a data structure of the same size in the shadow memory, which is divided into 16 four-byte entries corresponding to the 16 application words in the cache line. The first byte of the first entry is used as the bitmap for tracking the ownership of the entire cache line as before. The last two bytes of each entry are used as the read and write bitmaps to track access history for 8 threads for the associated word in the cache line. The second byte of each entry is not used. We use bit recycling to handle applications that need to create more than 8 threads.

We check whether there is a cache miss or invalidation using the ownership bitmap as described before. In addition, we set the bit corresponding to the word that was accessed by the current thread using an `OR` operation. We again use a test and set scheme to avoid unnecessary memory updates for better performance. On a cache miss, we check whether the word being accessed has write bits set by other threads. If not, a false sharing instance is reported. On a cache invalidation, we check whether any other thread had accessed the word by checking its bitmap. If there were no previous accesses, false sharing is reported. For a store, we clear the whole shadow memory unit for the cache line and set the cache line bitmap and the word bitmap with the write bit for the current thread. This approach limits multi-word updates to only cache invalidation events.

Our use of word granularity in false sharing detection may cause inaccuracy as some applications may have false sharing at the level of individual bytes. It is possible to implement byte granularity in our scheme. There are two reasons that we chose to use words instead. First, a byte granularity implementation incurs higher overhead. Second, if the compiler arranges for two unrelated data fields to lie in the same cache line, it is easier for the programmer to find that sub-word data's layout, e.g. an array of characters or several sub-word struct fields. In addition, the compiler often generates padding to expand unrelated adjacent sub-word fields to have word alignment. Thus we choose to use word granularity.

3.5 Accuracy Discussion

There are several potential sources of inaccuracy in our design and implementations. One source of inaccuracy is our assumption that data is only evicted from a core's private cache through invalidations. This may cause our profiling results to differ from the actual behavior of the application when executing on a particular hardware configuration, i.e., a particular private cache size and replacement scheme. However, our approach separates application behavior from the actual hardware configuration, which provides users with an insightful view of application behavior independent of any particular hardware features or configurations except the cache line size. We believe that this is more important than accurately reporting behavior on a particular execution instance, which may change when the application runs in a different environment.

In fact, the accuracy of cache contention analysis is difficult to evaluate. It depends on many factors, especially the interleaved order of memory accesses by different threads, which may change from run to run. Although full cache simulation can simulate detailed cache behavior, the input, i.e., the memory access trace, does not always reflect actual execution; thus, the simulation results are often artificial. For example, Cachegrind, a cache simulation tool built on Valgrind [21], serializes multithreaded execution, resulting in a memory reference sequence that would almost never happen in actual execution on a multi-core system. Even full cache simulation can report accurate results for one particular execution instance only and may not accurately reflect application behavior in other runs with different interleavings.

Other sources of inaccuracy, including the number of threads exceeding the number of bits in our bitmap, using word rather than byte granularity in false sharing detection, and not being able to separate cold misses from cache misses due to cache contention, have been discussed above.

4. Experimental Results

We conducted a series of experiments to evaluate our approach.

4.1 Experimental Setup

Our tool is implemented on top of Umbra for Linux. We used applications from two parallel, shared-memory benchmark suites for our experiments: the SPLASH2 [36] and Phoenix [27] suites. Three benchmarks from SPLASH2, namely `volrend`, `water-spatial`, and `cholesky`, were not included in our experiments because they complete in very short times (< 0.5 seconds). All benchmarks were compiled as 64-bit executables using gcc 4.3.2 with the `-O2` optimization flag. The hardware platform we used has two quad-core Intel Xeon processors with 3.16GHz clock rate and 8GB total RAM running 64-bit Debian GNU/Linux 5.0. Each core has its own private 32KB L1 data cache, whose cache line size is 64 bytes. The four cores in each quad-core die are arranged into two groups of two cores each that share a 6MB L2 cache. Thus, the fastest a given core can acquire a copy of a missing cache line is from the other core that shares its L2 cache, followed by one of the other cores on the same die. Fetching a cache line from the other die is even slower, followed by the case when a cache line must be loaded from memory.

4.2 Performance Evaluation

We first evaluate the performance of our base framework that tracks cache line ownership implemented using different instrumentation schemes: set only, test and set, and atomic test and set, which we discussed in Section 3. Table 2 shows the performance of the two benchmark suites with these different instrumentation schemes. It is clear that test and set has much lower runtime overhead. The reasons have been discussed in Section 3.1. The atomic test and set

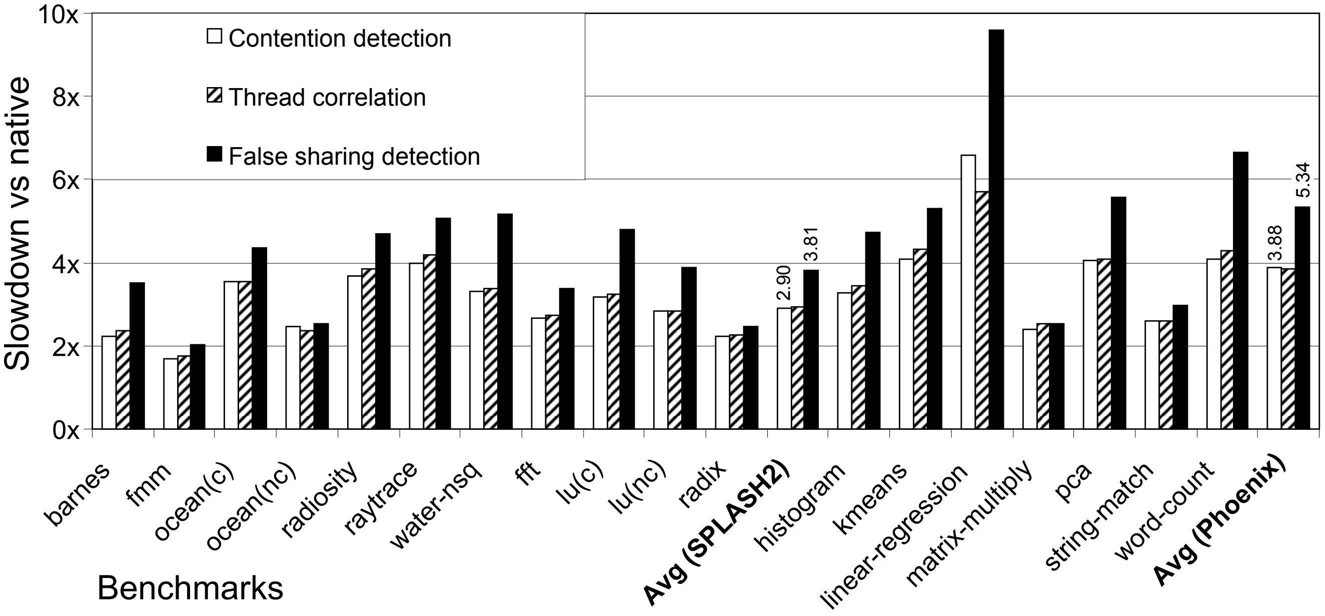

Figure 5. Performance evaluation of contention detection, thread correlation, and false sharing detection. Performance is normalized to 8-thread native execution time. Arithmetic average is used.

is much slower than the simple set scheme, primarily because of the high cost of the XCHG instruction.

Benchmarks	Set	Racy test & set	Atomic test & set
SPLASH2	31.83×	2.68×	50.82×
Phoenix	50.64×	3.61×	95.88×

Table 2. Performance of our base framework with different instrumentation schemes for tracking 8 threads, normalized to native execution (i.e., without any instrumentation) and averaged across each benchmark suite.

We next compare the performance of our three contention detectors. Figure 5 presents the results. It shows that contention detection and thread correlation have similar performance overhead. This is easily understood: they both check for changes in shadow memory bitmaps, and update any counters accordingly (see Section 3). Thread correlation is slightly slower than contention detection because it also needs to record the thread that last updated the cache line for each memory access. False sharing detection is the slowest. This is due mainly to the more complex instrumentation code needed to update the access positions within each cache line. On average, our tool causes a 3× to 5× slowdown when compared to native execution.

4.3 Accuracy of Racy Updates

Racy shadow memory checks and updates have much better performance than using atomic test and set in the instrumentation scheme on the shadow memory. However, races could result in loss of accuracy. Table 3 shows the relative errors (in percentages) of the miss and invalidation counts obtained using racy test and set versus atomic test and set instrumentation. We found that there are negligible differences between the two results for most benchmarks. For four of the 18 benchmarks, the error exceeded 10%. In our analysis, it is the relative magnitude rather than the absolute counts that matters. As such, we regard the trade-off between performance and accuracy of the final counts a worthwhile one to make.

Benchmark	Misses	Invalidation
`barnes`	0.15%	0.10%
`fmm`	20.52%	1.36%
`ocean (c)`	0.15%	0.13%
`ocean (nc)`	0.92%	0.55%
`radiosity`	10.94%	16.89%
`raytrace`	16.78%	22.06%
`water-nsq`	1.78%	2.54%
`fft`	0.00%	0.00%
`lu (c)`	0.23%	0.01%
`lu (nc)`	0.11%	0.02%
`radix`	0.00%	0.00%
`histogram`	0.00%	3.04%
`kmeans`	0.00%	0.00%
`linear_reg`	54.65%	69.91%
`matrix_mul`	0.13%	0.04%
`pca`	0.10%	0.67%
`string_match`	0.00%	2.82%
`word_count`	0.93%	0.17%

Table 3. Relative error between using racy versus atomic operations for counter updates.

Apart from the obvious reason that racy updates can cause inaccuracies, there is another possible cause for the differences: the slowdown may change how threads interleave their accesses, usually causing more interleaved accesses due to fewer references in each interval. An example is linear_reg: the atomic update causes higher runtime overhead, so more interleaved references happen, and more cache misses and invalidations occur and are detected. We believe this is the cause for the 50% variant of linear_reg in Table 3, not errors due to racy updates.

4.4 Benchmark Analysis

In this section, we describe some interesting benchmark results, and the insights they yielded. The way we analyzed the results also reflects how we believe a developer would use our tool to discover and fix inter-thread sharing contention problems.

4.4.1 Contention Detection

Contention detection can help programmers identify whether their multi-threaded programs suffer from cache contention due to data sharing, and pinpoint the most problematic instructions if there are any. The *contention rate* is the total number of cache misses and cache invalidations divided by the total number of instructions executed by all threads. Table 4 lists the speedups of our benchmarks due to parallelization using eight threads as well as their contention rates. Benchmarks with higher contention rates usually benefit little from parallelization. In other words, they are not scalable. The converse, however, is not true: a benchmark showing little speedup, as the number of threads and cores are scaled, does not necessarily have a high contention rate. There are many possible causes that limit the scalability of an application, e.g., imbalanced workload, expensive communication, or excessive synchronization. The rest of this paper focuses on those benchmarks that have high contention rates (i.e., a contention rate of more than 10^{-3}). These are the benchmarks ocean (nc), fft, histogram, radix, and linear_regression.

Benchmark	8-Thread Speedup	Contention rate
barnes	6.59	1.61×10^{-4}
fmm	6.42	8.36×10^{-5}
ocean (c)	4.26	6.12×10^{-4}
ocean (nc)	2.85	1.82×10^{-3}
radiosity	5.30	2.87×10^{-4}
raytrace	7.27	6.72×10^{-5}
water-nsq	5.87	2.34×10^{-5}
fft	2.07	2.28×10^{-3}
lu (c)	3.00	2.63×10^{-4}
lu (nc)	3.39	2.27×10^{-4}
radix	5.69	2.18×10^{-3}
histogram	3.18	4.26×10^{-3}
kmeans	3.96	1.09×10^{-5}
linear_regression	0.56	3.68×10^{-2}
matrix_multiply	1.02	6.85×10^{-5}
string_match	5.01	2.60×10^{-4}
word_count	7.24	1.32×10^{-4}

Table 4. The correlation between parallelization speedup and contention rate. The speedup is the native running time of a single thread divided by the total number of application instructions executed.

For the benchmarks with high contention rates, we examined the instructions causing the contention. There are four common reasons for their behavior:

1. Initialization. Some of the problematic instructions are found in initialization functions, especially in the first thread of the benchmark, which usually reads data from input files and causes cold misses. Because cold misses are treated as contention-causing misses for performance reason in our implementation (Section 3.2), such instructions are identified as delinquent accesses. For example, there are three instructions in histogram that are responsible for more than 99% of the cache misses, and all of them are from an initialization function that reads data.
2. Global data update. In some benchmarks, such as radiosity, threads update global data from time to time, which is often a source of cache contention.
3. False sharing. The linear_regression benchmark contains good examples of false sharing that resulted in many cache invalidations. It has 8 instructions that together are responsible for more than 99% of the total cache misses and invalidations. They all access data that resides in the same cache line from multiple threads. This will be discussed later in detail.
4. Communication. In some benchmarks, threads communicate with each other after each phase. fft is a typical benchmark that performs a lot of communication, thereby causing a high amount of contention.

4.4.2 Determining Thread Correlation

We next present results for our thread correlation algorithm. As expected, different benchmarks showed different communication patterns. The most common pattern we see is that all other threads have strong correlation with thread 0, the master thread. Benchmarks likes barnes, radiosity, lu, and water-nsquared have such patterns. In these benchmarks, the master thread will first initialize the data and then each thread works on its own tasks.

Some benchmark threads simply read data in parallel, and have little communication with each other. Most of the benchmarks from Phoenix fit this pattern. This is not surprising, since this set of benchmarks was designed to evaluate the performance of map-reduce, and so the benchmarks are mostly embarrassingly parallel.

We also tested the performance impact of scheduling different threads on different cores. In our platform, we have two quad cores on two different dies. Communication between two cores from different dies is much more expensive than communication between two cores on the same die. Based on the observed correlations, we cluster threads into two groups using two different methods. One method minimizes communication between the two groups while the other maximizes the communication. We considered the performance difference between these two methods on benchmarks with high contention rates based on thread correlations. histogram's correlation array shows that all threads interact most with the first entry, which indicates most misses are cold misess for first-time accesses. So it was not very interesting. fft and radix have almost identical correlation values among threads, revealing no optimization opportunities. ocean (nc) and linear_regression, on the other hand, showed interesting correlation patterns. ocean (nc) has a paired communication pattern: every two threads communicate significantly with each other but very little with the other threads. linear_regression, in contrast, forms a chain of communication. Each thread talks very frequently with its neighbors. As shown in Table 5, scheduling threads on different cores has a profound impact on the execution time for these two benchmarks. For ocean (nc), we observe a significant performance difference. The 'worst case', i.e., when threads were clustered to maximize communication, runs $2.4\times$ slower than the 'best case' where communication is minimized. The 'normal case', in which we left it to the operating system to decide the scheduling, is closer to the worst case. We see a smaller improvement in linear_regression, because it does not have a regular pattern that perfectly matches the core configuration as was the case in ocean (nc).

4.4.3 False Sharing Detection

Because software, including our benchmarks, is usually performance-tuned before release, many applications have already inserted padding on their own to avoid false sharing. It is therefore difficult

Benchmark	No binding	Worst case	Best case
ocean (nc)	1.95s	1.99s	0.79s
linear_regression	2.31s	3.08s	1.66s

Table 5. Performance impact of scheduling different threads on different cores for two of our benchmarks. All times are in seconds. 'No binding' lets the operating system decide the scheduling. The other two columns show the extremes of performance when scheduling on different cores. ocean's 'No binding' time averaged 1.95s but varied significantly.

Benchmark	Contention rate	False sharing rate
ocean (nc)	1.82×10^{-3}	1.57×10^{-3}
fft	2.28×10^{-3}	7.50×10^{-4}
radix	2.18×10^{-3}	1.16×10^{-3}
histogram	4.26×10^{-3}	4.26×10^{-3}
linear_regression	4.33×10^{-2}	4.33×10^{-2}

Table 6. False sharing rates of benchmarks with high contention rates. The false sharing rate is the number of false sharing instances divided by the total number of application instructions executed.

to find a mature program with significant performance problems. However, we did find several benchmarks with large amounts of false sharing. Table 6 shows the false sharing rate of the benchmarks that suffer from high contention rates. The *false sharing rate* is defined as the total number of false sharing misses divided by the total number of instructions executed. Our results show that these benchmarks also experience high rates of false sharing. The misses in histogram are mostly cold cache misses, which are identified as false sharing in our algorithm. It is possible to differentiate the cold misses from the real false sharing, but this requires extra instrumentation and shadow data fields. Note that the contention rate for linear_regression is different from the value in Table 4. This is because false sharing detection causes more runtime overhead than contention detection, and leads to more interleaved accesses on the same cache line by different threads, and thus higher contention rates.

By studying the source code of linear_regression, we found that it allocates an array, passes one entry to each thread, and each thread then updates the entry simultaneously. This is a common programming practice that causes false sharing. Because several entries share a cache line, a large amount of false sharing happens when multiple threads update neighboring entries. By adding pads into the data structure, we achieved a significant improvement (12× for 8 cores) for this benchmark, as shown in Figure 6.

We also find that the benchmark radix from SPLASH2 [36] has a significant amount of false sharing. radix is a benchmark that implements a parallel radix sort that sorts an array one digit at a time. In every round, it copies entries from the array into another array of the same size ordered by that round's digit. Multiple threads fill in the array, causing false sharing. However, this is behavior that we cannot change without drastically changing the algorithm. We tried to add padding between the data but the runtime actually increased because of lost cache locality. In fact, radix has a relatively good speed up, as shown in Table 4. This is because it sorts over 50 million integers, using two 400MB chunks of memory. Its working set is much larger than the available amount of caches in all eight cores. So there is a good chance that the data may already be evicted before invalidations or misses can happen.

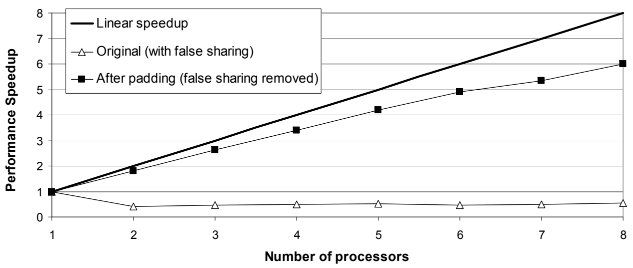

Figure 6. Speedup with and without false sharing. To eliminate false sharing we added padding to the data structures identified by our tool.

ocean (nc) also exhibited significant false sharing as well and has algorithmic issues. In ocean (nc), different threads update different columns of a 2D array. Because the array is stored in row-major order, a lot of false sharing occurs during this updating process. Our data shows that almost every instruction that accesses the array experiences false sharing. Due to the large size of the array, padding does not work well. However, as we showed in the previous section, we can still improve the overall performance by judiciously scheduling the threads so as to reduce communication cost. fft, in contrast, has an all-to-all communication pattern, so the majority of contention is actually caused by true sharing, and it has no easy way to optimize it due to its communication pattern.

An artifact of our implementation is that false sharing detection can precisely track up to 8 threads, while contention detection can track upto 32 threads. There are two benchmarks pca and kmeans that use more than 8 threads. They create 8 threads in every phases or iteration. In total, pca and kmeans used 16 and 1,317 threads respectively. Table 4.4.3 lists the data obtained from contention detection and false sharing detection. False sharing detection reported less cache misses and invalidations because fewer number of bits were used in thread tracking. How to precisely and efficiently track a large number of threads (hundreds or thousands) simultaneously is still an unresolved issue.

Benchmarks	Contention detection		False sharing detection	
	# misses	# inv	# misses	# inv
kmeans	3.23×10^{6}	1.53×10^{6}	4.25×10^{5}	4.25×10^{5}
pca	5.63×10^{6}	4.37×10^{6}	1.16×10^{6}	1.16×10^{6}

Table 7. The number of cache misses and invalidations observed during contention detection and false sharing detection for benchmarks that created more than 8 threads.

Our experiments showed that the novel analysis made possible by our tool can help programmers discover intense thread contention, determine thread correlation, and detect false sharing. This helps programmers better understand their applications and make better choices, for example, in the implementation of the data structures, or the scheduling decisions, so as to improve performance.

5. Related Work

5.1 False Sharing

False sharing can be difficult to define precisely [3, 18, 35]. For instance, false sharing can be defined as the additional cache misses incurred by a program running with a given cache line size compared to the same program running with one-word cache lines. This may seem like a good definition because using one-word cache lines minimizes the amount of data transferred between processors. However, since programs cannot exploit spatial locality with word-sized cache lines, the number of coherence operations needed between processors may still increase. Thus, this definition can result in a negative amount of false sharing if the loss in spatial locality eclipses any savings due to reduced false sharing. In this paper, we chose an intuitive definition of false sharing that can be practically used for dynamic detection.

Before the proliferation of multi-core systems, false sharing emerged as a significant performance problem for distributed shared memory (DSM) systems. However, because cache coherency protocols in DSM systems operate at the granularity of individual memory pages, DSM false sharing is much more likely to occur than cache line false sharing in multi-core machines. Many approaches developed to dynamically control DSM false sharing [7, 11] used relaxed memory consistency models and version vectors with smaller granularity than memory pages.

Intel's Performance Tuning Utility (PTU) [14] provides hints that can be used by developers to identify false sharing: for each cache line, it collects the thread ID and offset of sampled accesses. Also, the Precise Event Based Sampling (PEBS) support on Intel processors can provide reports of coherency events and identify addresses of the corresponding memory accesses. However, because both of these approaches are sampling-based, they only aggregate memory access statistics without recording the order of interleaved memory accesses. Thus, these approaches cannot distinguish between false and true sharing and can greatly overstate the incidence of false sharing. Hardware approaches [9] for reducing false sharing include protocols that perform invalidations on a word basis or postpone invalidations at the sender/receiver or both. However, such approaches rely on special hardware support.

Compiler optimizations for reducing both DSM and cache line false sharing have also been proposed in the past. Static analysis can be used to approximate program memory-access patterns and apply data layout transformations to improve memory locality [16]. Other proposed approaches reorganize control structures (e.g., loop distribution) along with shared data to reduce false sharing [17, 24]. These approaches rely on the regularity of code and data layout to approximate the memory reference behavior of programs, which greatly limits their usage and accuracy.

Memory managers such as Hoard [2] try to *prevent* false sharing caused by concurrent requests by ensuring that data allocated for separate threads does not lie on the same cache line. Unfortunately, these allocators have no control over inter-thread contention inadvertently caused by developers in real-world applications because of poor data layout or thread scheduling. Diagnostic tools are invariably required to help developers reduce false sharing and thread contention.

A full architecture simulation for detecting false sharing is employed by CacheIn [34]. The runtime overhead for the full simulation is not mentioned but we can reasonably expect it to be very high. Furthermore, the false sharing detection algorithm works by creating a serial trace of all memory references and comparing the address of shared writes to subsequent shared reads. This approach is not efficient in terms of the sheer amount of data generated during program execution and the likely post-processing overhead. Furthermore, the machine model of this simulation only uses the latencies of a few instructions. Traditional full simulations [15] can be used to obtain detailed information about cache behavior, but at the cost of orders of magnitude of slowdown. Our method is an order of magnitude faster.

Pluto [12] tries to detect cache false sharing via dynamic binary instrumentation. However, it simply aggregates information about the number of threads that access a given cache block (and the corresponding access offsets), without retaining any information about the timing of these accesses. While Pluto uses heuristics about thread offsets (just like PEBS) to account for true sharing, it cannot accurately differentiate between true or false sharing, and its results can be very inaccurate. It cannot accurately provide the cache invalidation or false sharing statistics for each instruction. Furthermore, Pluto's reported performance overhead can be as high as two orders of magnitude on target applications.

5.2 Thread Correlation

Most thread libraries allow programmers to specify thread relationships, execution priorities, and scheduling policies. For instance, programmers can statically bind threads to specific processors on most operating systems. Some frameworks allow programmers to declare groups of related threads (e.g., via RTIDs [26]) so that thread schedulers can run these related threads accordingly to avoid performance penalties due to thread communication.

Real-world applications are often developed in different phases, utilize modules from many libraries, and are coded by many programmers. Therefore, it may not be easy for application developers to specify the scheduling of threads or their processor bindings. In the absence of programmer directives, many frameworks track the cache behavior of threads at runtime to make scheduling decisions that improve performance. For instance, cache-aware schedulers can dynamically identify threads that reference shared data [6, 10, 32, 33] so that they can be scheduled accordingly or migrated to specific processors. Such optimizations can reduce overhead due to poor thread scheduling, but they often use hardware sampling of cache misses or other hardware facilities, which makes their approaches inaccurate and platform specific. As a result, these approaches often cannot help developers identify the exact threads and instructions in an application that communicate excessively with each other. With such diagnostic information, developers would be able to take more aggressive, application-specific steps to overcome thread correlation penalties. To the best of our knowledge, no existing dynamic instrumentation tool provides thread correlation analysis for diagnosis without incurring prohibitive performance overheads.

6. Conclusion

Merely porting an application to use multiple threads is insufficient to guarantee good performance on today's multi-core systems. In this paper, we focused on reducing unexpected performance degradation in multi-threaded applications that can arise from inter-thread cache contention and sub-optimal placement of correlated threads across multiple cores. In particular, we outlined a novel approach for identifying delinquent accesses, measuring thread correlation, and detecting true/false sharing. Our tool is based on the insight that the rate of true/false sharing in an application primarily depends on application behavior and cache line size, and can be accurately determined without considering the full complexity of the actual memory hierarchy. We outlined a novel use of shadow memory and dynamic instrumentation that tracks ownership of cache lines in application memory to detect true/false sharing. Our approach is more accurate than static analysis or the use of hardware counters, because we use the exact sequence of memory references in an executing application to detect both true and false sharing, and to differentiate between them. We show that our approach incurs an average of 3× to 5× slowdown relative to native execution, and thus is much more efficient than cache simulations.

We used our tool to analyze 18 shared memory benchmarks. We showed how the information obtained by our tool can be used to significantly improve application performance. We first performed contention detection to identify benchmarks that suffer from contention issues. We then examined thread correlation to derive the communication patterns of problematic applications. In one instance, namely ocean, we attained a 2.4× performance improvement by optimally scheduling the threads of the application. Next, we illustrated that applications with high contention rates consistently exhibit high degrees of false sharing. For one benchmark, namely linear_regression, we removed false sharing by means of padding and turned slowdowns originally experienced by the application into near-linear speedups: we turned a 2× slowdown into a 6× speedup relative to native execution on 8 cores. For other applications, the information that our tool provided was used to explain why no performance improvement was possible without drastic restructuring of the underlying algorithms.

In the future, we would like to explore how we can further speed up our approach through the use of *sampling*. If this turns out to be feasible, it may be possible to extend our tool into a runtime framework that interacts directly with the thread scheduler, allowing for tuning of multi-threaded applications as they execute through their various program phases, and dynamic load-balancing of the system. Alternatively, we could integrate our tool with micro-simulations [39] to obtain even greater detail about the application's and/or the platform's behavior.

References

[1] DynamoRIO dynamic instrumentation tool platform, Feb. 2009. http://dynamorio.org/.

[2] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM SIGPLAN Notices*, 35(11):117–128, 2000.

[3] P. W. Bolosky, W. J. Bolosky, and M. L. Scott. False sharing and its effect on shared memory. In *In Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 57–71, 1993.

[4] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, M.I.T., Sept. 2004.

[5] M. Burrows, S. N. Freund, and J. L. Wiener. Run-time type checking for binary programs. In *Proc. of the 12th International Conference on Compiler Construction (CC '03)*, pages 90–105, 2003.

[6] J. M. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. *Real-Time Systems, Euromicro Conference on*, 0:194–204, 2009.

[7] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, page 164. ACM, 1991.

[8] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proc. of the Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC '06)*, pages 749–754, 2006.

[9] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom. The detection and elimination of useless misses in multiprocessors. *ACM SIGARCH Computer Architecture News*, 21(2):88–97, 1993.

[10] A. Fedorova. *Operating system scheduling for chip multithreaded processors*. PhD thesis, Harvard University, Cambridge, MA, USA, 2006.

[11] V. W. Freeh. Dynamically controlling false sharing in distributed shared memory. *International Symposium on High-Performance Distributed Computing*, 0:403, 1996.

[12] S. Gunther and J. Weidendorfer. Assessing cache false sharing effects by dynamic binary instrumentation. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 26–33. ACM, 2009.

[13] J. J. Harrow. Runtime checking of multithreaded applications with visual threads. In *Proc. of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 331–342, 2000.

[14] Intel-Corporation. Intel Performance Tuning Utility 3.2. *User Guide, Chapter 7.4.6.5*, 2008.

[15] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A Pin-based on-the-fly multi-core cache simulator. In *Proc. of the The Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, pages 28–36, Beijing, China, Jun 2008.

[16] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. *ACM SIGPLAN Notices*, 30(8):179–188, 1995.

[17] Y. Ju and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. *Languages and Compilers for Parallel Computing*, pages 344–358, 1992.

[18] V. Khera, P. R. LaRowe, Jr., and S. C. Ellis. An architecture-independent analysis of false sharing. Technical Report DUKE-TR-1993-13, Duke University, Durham, NC, USA, 1993.

[19] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *Proc. of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '06/Performance '06)*, pages 216–227, 2006.

[20] N. Nethercote and A. Mycroft. Redux: A dynamic dataflow tracer. In *Electronic Notes in Theoretical Computer Science*, volume 89, 2003.

[21] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 89–100, June 2007.

[22] J. Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[23] OpenWorks LLP. Helgrind: A data race detector, 2007. http://valgrind.org/docs/manual/hg-manual.html/.

[24] J. Peir and R. Cytron. Minimum distance: A method for partitioning recurrences for multiprocessors. *IEEE Transactions on Computers*, 38(8):1203–1211, 1989.

[25] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. of the 39th International Symposium on Microarchitecture (MICRO 39)*, pages 135–148, 2006.

[26] M. Rajagopalan, B. Lewis, and T. Anderson. Thread scheduling for multi-core platforms. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6. USENIX Association, 2007.

[27] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[28] Rational Software. Purify: Fast detection of memory leaks and access errors, 2000. http://www.rationalsoftware.com/products/whitepapers/319.jsp.

[29] M. Ronsse, B. Stougie, J. Maebe, F. Cornelis, and K. D. Bosschere. An efficient data race detector backend for DIOTA. In *Parallel Computing: Software Technology, Algorithms, Architectures & Applications*, volume 13, pages 39–46. Elsevier, 2 2004.

[30] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[31] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proc. of the USENIX Annual Technical Conference*, pages 2–2, 2005.

[32] S. Sridharan, B. Keck, R. Murphy, S. Chandra, and P. Kogge. Thread migration to improve synchronization performance. In *Workshop on Operating System Interference in High Performance Applications*, 2006.

[33] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 47–58, New York, NY, USA, 2007. ACM.

[34] J. Tao and W. Karl. CacheIn: A Toolset for Comprehensive Cache Inspection. *Computational Science–ICCS 2005*, pages 174–181, 2005.

[35] J. Weidendorfer, M. Ott, T. Klug, and C. Trinitis. Latencies of conflicting writes on contemporary multicore architectures. *Parallel Computing Technologies*, pages 318–327, 2007.

[36] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proc. of the 22nd International Symposium on Computer Architecture (ISCA '95)*, pages 24–36, 1995.

[37] Q. Zhao, D. Bruening, and S. Amarasinghe. Efficient memory shadowing for 64-bit architectures. In *Proc. of the The International Symposium on Memory Management (ISMM '10)*, Toronto, Canada, Jun 2010.

[38] Q. Zhao, D. Bruening, and S. Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *Proc. of the International Symposium on Code Generation and Optimization (CGO '10)*, Apr. 2010.

[39] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. Ubiquitous memory introspection. In *International Symposium on Code Generation and Optimization*, San Jose, CA, Mar 2007.

[40] Q. Zhao, R. M. Rabbah, S. P. Amarasinghe, L. Rudolph, and W.-F. Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Proc. of the 17th International Conference on Compiler Construction (CC '08)*, pages 147–162, 2008.