COMPILER COST MODEL FOR MULTICORE ARCHITECTURES

A Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Munara Tolubaeva

May 2014

COMPILER COST MODEL FOR MULTICORE ARCHITECTURES

Munara Tolubaeva

APPROVED:

Dr. Barbara Chapman, Chairman
Dept. of Computer Science, University of Houston

Dr. Edgar Gabriel
Dept. of Computer Science, University of Houston

Dr. Weidong Shi
Dept. of Computer Science, University of Houston

Dr. Yonghong Yan
Dept. of Computer Science, University of Houston

Dr. Hakduran Koc
Dept. of Computer Engineering,
University of Houston - Clear Lake

Dean, College of Natural Sciences and Mathematics

A C K N O W L E D G M E N T S

I would like to express profound gratitude to my adviser Professor Barbara Chapman for all guidance, advice, and support that she has given me during my Ph.D. studies. I have learned a lot from her which will benefit me for my whole life.

My special thanks go to Dr. Yonghong Yan for his guidance, his insightful comments, and his support and understanding throughout this endeavor.

I would like to express my appreciation to my dissertation committee for their valuable suggestions.

I want to thank God for giving me such beautiful (in all its comprehension) parents, my father Jolchu Tolubaev and my mother Inabat Tolubaeva. Without them I would not be standing here and writing this dissertation. I am very grateful for my father who has made the most crucial choice in my whole life by sending me to the best high school at that time which serves as a start of all my accomplishments and career dreams so far. He has always supported me throughout my success and failures on this challenging, hard working path. I also want to thank my dearest mother for always supporting me and being by my side both on good and bad days in my life.

Lastly, I would like to give a special thanks to my lovely husband, Emil Bilgazyev, for being very supportive and patient during the course of the Ph.D. studies. I am very happy and feel lucky to have him as my husband.

COMPILER COST MODEL FOR MULTICORE ARCHITECTURES

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Munara Tolubaeva

May 2014

Abstract

The intention to move from single core to multicore architectures has been to increase the performance of a system and hence increase the performance of an application. However, obtaining the optimal application performance on multicore architectures is found to be not that trivial and still remains as unsolved problem due to the multiple challenges the multicore architectures face. The main reason for all the challenges that the multicore systems face is the inability to utilize the system resources well enough. Ineffective utilization or poor coordination of resources may create performance bottlenecks and overheads on the system that ultimately affects the overall performance of an application. We have identified three main causes of performance degradation on multicore architectures; these are false sharing, memory bandwidth, and shared last level cache contention. Knowing the degree to which an application performance would degrade due to these three issues would give an idea to an application programmer or compiler as to which code transformation is needed in order to decrease this negative performance impact. Unfortunately, the current state-of-the-art compilers such as Open64 and GNU are oblivious to these performance bottlenecks stated above. Even though these compilers, especially Open64, have a very robust optimization and code transformation phases, they are all limited to sequential programs and simple architectures with single processor units. This limitation makes their optimization phases less accurate on multicore architectures. In order to improve compilers' code transformation and optimization phases, compilers' cost models that guide optimizations should be extended to consider these performance bottlenecks that can occur on multicore architectures. Therefore, the

goal of this dissertation is to develop compile time models that quantitatively estimate the impact caused from these three performance degrading bottlenecks to the overall application performance, and that can be used as extensions to the existing compilers' cost models when guiding certain optimizations and/or code transformations targeting multicore architectures.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Compiler cost models [60], or performance models, are analytical models to estimate the cost of executing a specific section of code, such as computation-intensive loops. It is often used in the compiler to decide whether certain optimization has performance benefits, thus to guide the optimization. For example, in an optimizing compiler, loop nest optimizations (LNO) such as loop interchange, tiling, and unrolling, are widely used techniques for improving the performance of loops. Compiler transformations in LNO phase improve spatial locality of loops by changing the loop structures and the order of iterations. The locality and performance benefits from LNO depend largely on the parameters of loop transformations such as the unrolling factor, the tile size. Poorly chosen parameters will degrade the performance of a loop.

One approach for choosing good parameters is to use a cost model. Before applying a specific transformation with certain parameters, the compiler uses analytical

models to estimate the costs of executing the loops in its original version and in the transformed version. The compiler then decides whether the transformation is beneficial or not by comparing the two costs. The costs are often calculated as CPU cycles needed to execute two versions of the loop, considering hardware architectures and software environment that affect the loop performance, such as cache organization, processor frequency, and runtime overhead.

Multicore processors have become ubiquitous and are used by many different users, ranging from inexperienced programmers to professionals and scientists. Parallel programming, once considered for the high-end computing, is now becoming a common practice and required expertise for average programmers. Multithreaded programming APIs such as OpenMP [8] provide a productive programming environment for creating parallel programs from the sequential versions. However, obtaining scalable performance on large parallel machines still requires significant amount of effort in performance tuning, especially with regards to data locality. It becomes necessary for programmers to understand concepts such as caches and locality, data and work sharing, and synchronizations in order to write parallel programs that will deliver good performance.

The only drawback of compiler cost models is that they target single processor architectures, and are not very useful for state-of-art architectures such as multicore and many-core systems. Multicore and manycore computer architectures include various hardware resources that are shared by processing cores, e.g., shared cache, memory bandwidth, and interconnects. Efficient use of these shared resources is crucial to the overall system and application performance. It requires both to maximize

the sharing of these resources among concurrent threads, and also to minimize the contentions and conflicts of using them. Existing compiler cost models do not take into account the performance impact from the interference or contention for these shared resources such as false sharing effects, the competition to use shared cache or memory bus. On these systems, shared resource interference and contention will have significant performance impacts for applications. Thus, it is very important for compilers' and performance estimating tools' cost models to accurately estimate the execution performance of applications on these architectures by considering concurrent utilization of limited shared resources.

1.1 Research Goals and Contributions

The goal of this work is to extend existing compiler cost models by defining new compile time models for data parallel applications with regards to the impact of contention caused by using shared resources on current many-core and multicore platforms, including the shared cache and memory bandwidth.

In data parallel OpenMP applications, OpenMP loops play a significant role for overall performance of the program. By analyzing the concurrent execution of iterations of the loop by threads and by knowing the layout of the underlying architecture, new cost models will be used to predict additional CPU cycles needed to execute the loop due to multithreading. This information will be used by compilers to guide the parallel loop transformations by obtaining more accurate timing estimations for the loops. Compilers will also be able to use information from models to perform

automatic optimizations, such as changing the loop iteration behavior or data alignment to eliminate or minimize these negative performance impacts. These modeling and estimation results could also be useful for programmers for performance tuning and locality optimizations. Runtime systems could use information from new cost models for runtime dynamic or feedback-driven optimizations. The quantitative performance impact information will be especially helpful when tuning an application for specific hardware architectures.

Our work makes the following contributions:

1. Build a model to output the total number of false sharing cases that will occur during execution of the parallel loop and to analyze the performance impact of false sharing on a parallel loop as a percentage of execution time.

2. Introduce a linear regression model to reduce the false sharing modeling time by approximation without impacting its accuracy.

3. Introduce a modified STREAM kernel that is used with the curve fitting technique to derive the statistical memory bandwidth model for a particular system with regards to the parallelism and concurrent cache misses.

4. Build compile time statistical model that can be used to predict the memory bandwidth requirement of parallel loops when being executed with specific number of threads.

5. Introduce a new method to predict number of cache misses that would happen due to cache sharing and/or contention when multiple threads are co-scheduled

to execute simultaneously.

1.2 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 explains the general motivation behind this dissertation. Chapter 3 presents background information about Open64 compiler and its cost models. Chapter 4 describes the false sharing concept and explains our model of estimating the false sharing impact on a parallel loop. Off-chip memory bandwidth modeling is presented in Chapter 5. Chapter 6 explains the details of modeling cache contention and/or sharing impact at compile time. Lastly, we conclude the dissertation with our findings in Chapter 7.

Chapter 2

Motivation

The evolution of multicore processors has started due to the growing demand for faster computing performance. A solution for improving the performance of single core processors was to increase the processor frequency which would enable the processor to execute instructions in much quicker time. This trend was followed only until 2002 because system designers hit a *power wall* i.e. when processor's power consumption increases exponentially with each increase of the processor frequency [55]. In order to overcome this limitation of single core processors, vendors have switched the trend to produce multicore processors where multiple less powerful processors are combined on the same die instead of having a single high performing processor. Multicore processors have proven to boost the application performance over single cores [24, 25, 65, 55]. The reason for this performance boost is not that the cores on multicore processor are clocked at a higher frequency rate, but instead is the ability to execute multiple programs independently and concurrently which ultimately

contributes to the overall performance.

In spite of the many advantages of multicore processors, obtaining the maximum performance out of an application and the system still remains a big challenge. This is due to the fact that the difference between single core and multicore processors is not only in the number of cores they possess, but also in memory hierarchy, the interaction between on-chip components such as dedicated caches, and presence of commonly shared components [52]. Assuming that a sequential program obtains X performance when executed on one of the cores of multicore architecture, one would be mistaken to expect $4X$ performance by running four different sequential programs concurrently on four cores of the architecture. This is because of scalability issues that multicore processors face. The main cause of poor scalability on multicore processors is the inability to utilize the resources on the chip well enough. Ineffective coordination of resources may create performance bottlenecks and overheads on the system that ultimately affects the overall performance of an application. Thus, the most important question one needs to ask is how to rearrange an application or in which configuration to run the application so that the application achieves full performance on multicore processor.

Modifying an application code and/or finding the best configuration to run the application with may be needed to prevent performance degrading bottlenecks that are caused by either poorly coordinated accesses to both private and shared resources or over-demanding the already limited amount of shared resources on multicore architectures. This is the main reason why there has been so many studies focusing on identification of the causes of system bottlenecks and estimation of the degree of

the effect it would cause on the application performance [25, 55, 28, 14]. Memory controller, memory bus, and last level cache are all identified as causes of performance degradation on multicore architectures and have been thoroughly studied in the literature [65, 28, 14]. Knowing the degree to which an application performance would degrade quantitatively due to poor utilization of these resources would give an idea to an application programmer or compiler to perform some kind of code transformations that would decrease this negative performance impact.

In addition to performing code rearrangements to decrease the negative performance impact, one could decide on the optimal execution configuration to run the application with, such as deciding on the optimal number of threads, placement of threads on cores, decision of how a multithreaded or multi-tasking application be distributed to threads, or how to optimally schedule independent tasks of an application in order to avoid performance bottlenecks and achieve the highest performance. These decisions are hard to make unless one has comparable results from different execution configurations. Therefore, it would be useful for an application programmer and/or compiler to know approximate estimations regarding the performance impact a certain bottleneck would cause. Based on these estimations, they would make a decision of how to modify/rearrange the code or how to run it in order to prevent the bottleneck and ultimately get rid of the negative performance impact.

Authors of [14] studied traditional single core measurements that guide compiler optimizations, predict application performance, and showed how these measurements can be deceptive in a multicore based systems. Their work demonstrates that analyses and optimizations to identify and mitigate performance bottlenecks on single

core architectures are inadequate for multicore architecture [17]. Moreover, they state that the same optimization can give different outcomes in single core and multicore architectures. For example, an optimization that increases performance on single core may decrease it on multicore system and vice versa.

In single core architecture, L1 and L2 cache miss ratios provide an idea about the degree of a memory bottleneck. However, on multicore architecture, concluding about the memory bottleneck due to high L1 and L2 cache miss ratios won't be quite accurate because of the other factors, such as last level cache miss ratios that should be taken into account. On the other hand, an application may exhibit great L1 performance due to the hardware/software prefetching, but severely suffer from lower memory bottlenecks [14]. Therefore, this study also proves the necessity of modeling shared resources on multicore processors in order to accurately estimate the application performance and/or performance bottleneck.

Overall, obtaining a good application performance on multicore architecture is not a trivial job, because one has to effectively utilize all the private and shared resources on the system in order to eliminate any system bottleneck from happening. Therefore, there have been many different studies in the literature that have focused on helping an application programmer to prevent a bottleneck or to analyze how well a certain resource is being used by an application.

Quite a large amount of work has been done in investigating contention aware scheduling techniques that would decrease the causes of performance degradation due to contention for shared resources on multicore processors [65, 3, 41, 13]. These studies have identified various shared resources that would cause contention, such as

memory controller, memory bus, and prefetching hardware, as well as last level cache, and proposed solutions on the optimal thread scheduling techniques to mitigate the contention as much as possible.

Memory bus or bandwidth contention has also been agreed by many studies to cause performance bottlenecks on multicore architectures [28, 14, 24, 43]. In [14] authors in fact show that memory bandwidth contention is the primary source of damage to the application performance when moving from single to multicore based systems. In order to eliminate the memory bandwidth contention, the authors suggest application developers understand that the high percentage of peak performance can be achieved only by arranging the data to be cache friendly. In addition to memory bus contention, several other studies have identified shared last level cache capacity as one of the performance bottlenecks in multicore based systems [32, 58, 47, 63, 66, 7]. Some of these works focused on finding techniques to reduce the last level cache contention using different scheduling order [58], *Utility Page Partitioning* [47] and *Page Coloring* [63]. The application performance may also suffer from high cache coherency overhead on multicore processors [24, 25, 45]. Cache coherency overhead is caused due to high false sharing cases that happen throughout the program execution. Mitigating or eliminating false sharing has been widely studied as well in the literature.

In this dissertation, we focus on three concepts that have been commonly accepted by many existing research works (as discussed above) to cause performance degradation on multicore architectures which are false sharing overhead, memory bandwidth and last level cache contention. The very broad contribution of the dissertation is

to model these three issues at compile time for OpenMP parallel loops, and quantitatively analyze how these issues impact the overall performance of parallel loops. We believe that the information provided by our models can be extremely useful for application programmers, compilers, and performance tuning tools in deciding or guiding code transformations and optimizations that would mitigate the possible performance impact from these three bottlenecks on multicore architectures.

Chapter 3

Background

In this chapter, we introduce Open64 compiler [60] and the cost models used in the compiler to drive loop nest optimizations.

3.1 Open64 Compiler

Open64 compiler suite is an open source state-of-art compiler that supports programs in Fortran 90/95, C and C++ including OpenMP directives. The compiler consists of several independent modules that interact via a common intermediate representation (IR) called WHIRL as shown in Figure 3.1. There are different levels of IR that are being used by the modules during compilation. Front-end of the compiler generates very high-level IR from the input source code, which is used as an input to the Very High-Level Optimizer. Back-end of the compiler is divided into interprocedural analysis and optimization, loop nest optimization, global optimization and

code generation components where each of these components take a different level of IR as an input. The interprocedural analysis performs symbol analysis, constant global identification, array sections, and code layout for locality to gather information about the code. Optimizations such as procedure inlining, intrinsic function inlining, cloning for constants, dead function and dead variable elimination and interprocedural constant propagation are then applied based on the interprocedural analysis. The loop nest optimization (LNO) phase performs data dependence analysis to check if there is any data dependence issue in a loop. At this phase, compiler cost models determine what transformations should be applied to the loop such as loop fission, loop fusion, loop interchange, loop tiling, loop peeling, loop unrolling. If desired, automatic parallelization with OpenMP directives is performed as well. The global optimization phase creates control flow graph (CFG), converts IR to static single assignment (SSA) form, performs dead code elimination, dead store elimination, copy propagation, SSA pointer alias analysis, CFG optimization, strength reduction and aggressive code motion on SSA form, and translates back to IR after optimizations are performed. The code generation phase performs software pipelining, global code motion between basic blocks, and peephole optimizations.

3.2 Cost Models

As discussed previously, Open64 compiler uses a set of cost models to decide on what transformations to perform on a loop during the LNO phase. In this section, we introduce these cost models in more details and explain their responsibilities

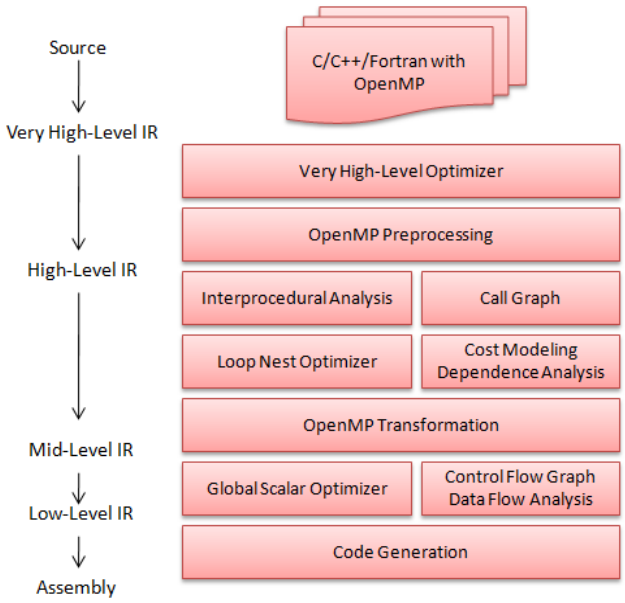


Figure 3.1: The components of the Open64 compiler.

during the LNO phase. There are three cost models in Open64 compiler which are *Processor, Cache,* and *Parallel* models.

3.2.1 Processor Model

The processor model estimates the time, in CPU cycles, needed to execute one iteration of the loop ($Machine_c\_per\_iter$) by modeling computational resources, registers, and operation latencies, as shown in Equation 3.1. The model tries to predict the scheduling of instructions ($Resource_c$) given the available amount of resources such as arithmetic/floating point units, memory and issue units. It considers the dependencies ($Dependency\_latency_c$) among instruction/memory operations and estimates the processor stalls caused by latencies. The Open64 compiler uses the processor model to make decisions regarding the best loop unrolling factor to apply to the loop.

$$
\begin{aligned}
&Machine_c\_per\_iter = Resource_c + Dependency\_latency_c + Register\_spilling_c \\
&Resource_c = maximum(Op_c, MEM\_ref_c, Issue_c) \\
&MEM\_ref_c = (Num\_fp\_refs + Num\_int\_refs)/Num\_mem\_units \\
&Issue_c = Num\_inst/Issue\_rate \\
&Dependency\_latency_c = maximum(Sum\_of\_latencies_i/Sum\_of\_distances_i) \\
&Register\_spilling_c = (Reg\_used - Target\_regs) * \\
&\qquad\qquad (Num\_reg\_refs/(Scalar\_regs + Array\_regs)) \\
&Reg\_used = Base\_regs + Scalar\_regs + Array\_regs
\end{aligned}
\tag{3.1}
$$

3.2.2 Cache Model

The cache model predicts the number of cache misses and estimates additional cycles needed to execute one iteration of an inner loop. In addition, the model is responsible for identifying the best possible loop block size for loop tiling. The number of cache misses is determined by summing up the footprints at the loop level [60]. The footprint is the number of bytes of single data reference in a cache. Due to spatial data locality, footprints of consecutive array references are counted only once. For example, references a[i] and a[i+1] lie in the same reference group, thus have only one footprint. When there is a reference to a[i], the cache line containing a[i] would be placed in a cache. The unused data in the same cache line would also be considered when counting the footprints. When the total amount of footprints is gathered, the model compares whether the footprint size is larger than the cache size; if so, a cache miss is considered to occur.

Equation 3.2 shows the equations of the cache model. The Translation Look-aside Buffer (TLB) cost is computed in the same way as the cache cost, because the TLB is modeled as another level of cache.

$$
\begin{aligned}
&if(Num\_array\_ref - TLB\_entries > 0) \\
&\qquad TLB\_miss = Num\_array\_ref - TLB\_entries \\
&TLB_c = TLB\_miss\_penalty * TLB\_miss \\
&Cache_c = \textstyle\sum_i^{Levels}(Clean\_footprint_i * Clean\_penalty_i \\
&\qquad + Dirty\_footprint_i * Dirty\_penalty_i)
\end{aligned}
\tag{3.2}
$$

3.2.3 Parallel Model

The parallel model helps the compiler to decide whether the parallelization of a loop is possible and if so which loop level is the best candidate for parallelization. The model makes use of the *Processor* and *Cache* models that are discussed in the previous sections, and also considers the cost of loop and parallel overheads as shown in Equation 3.3. The loop overhead considers the time needed to increment the loop indices, to check the loop boundary condition for each iteration which are aggregated into $Loop\_overhead\_per\_iter_i$. The parallel overhead is the time taken to actually execute the parallel loop. It includes overheads due to parallel startup, scheduling iterations, synchronizations and worksharing between threads [29]. In this work, the parallel overhead will refer to the OpenMP overhead incurred when parallelizing the loop via OpenMP constructs.

$$
\begin{aligned}
Total_c &= Machine_c + TLB_c + Cache_c + Loop\_Overhead_c + Parallel\_Overhead_c \\
Machine_c &= Machine_c\_per\_iter * Num\_loop\_iter/Num\_threads \\
Loop\_overhead_c &= Loop\_overhead\_per\_iter_i * Num\_loop\_iter/Num\_threads \\
Parallel\_overhead_c &= Parallel\_startup_c + Parallel\_const\_factor_c * Num\_threads
\end{aligned}
\tag{3.3}
$$

As discussed in Sections 3.2.1, 3.2.2, and 3.2.3, the *Processor model* evaluates the resource utilization cost, the *Cache model* analyzes the spatial and temporal locality of single thread execution, while the *Parallel model* focuses on the worksharing benefits from concurrent execution of threads. However, neither of them, nor their combination takes into account the performance impact from the interference or contention for resources between parallel threads such as the false sharing effects, the competition to use shared cache or memory bus. With increasing number of cores and the decrease of average memory and bus bandwidth per cores as in multicore and many-core architectures, such interference and contention will have significant performance impacts for applications. Therefore, it is very important for compilers' and performance estimating tools' cost models to accurately estimate the execution performance of applications on these architectures by considering concurrent utilization of limited shared resources.

Chapter 4

False Sharing Modeling

In this section, we first explain the false sharing concept and its performance impact on parallel applications. We then discuss our approach of modeling false sharing impact at compile time. Lastly, we present our model validation results.

4.1 Introduction

False sharing (FS) occurs when two or more threads access the same cache line, but different elements of the line and at least one of these accesses is a write operation [4]. For example, consider two threads A and B executing on two distinct cores of a parallel system where cache coherency is maintained by a write-invalidate protocol. If the threads perform memory references that cause the processor to fetch the same cache line to their private caches, and when one thread modifies some element on the cache line, the state-of-the line stored in the other thread's private cache becomes

invalid due to the protocol. Right after the invalidation, when the second thread accesses any element of the invalidated cache line, a cache miss occurs. The second thread will unnecessarily have to fetch the line from memory, even if the requested elements were not modified. If FS happens frequently during the execution of the program, then the overhead caused by the problem will significantly decrease the overall performance of the program.

The FS problem is a well-known performance degrading issue in parallel programs, whose performance impact when executing a victim[1] loop may be as astonishing high as 60% [59]. It occurs at the cache line granularity and closer to the architecture level; detection of it is not obvious for both average and experienced programmers. It is related to private caches and the cache coherency protocol that enforces a consistent view of the memory by all private caches, concepts that are often hard to understand for average programmers. FS reflects performance degrading data access pattern, but can only reveal such pattern after program execution. It is often hidden from programmers' view during the program creation and it is hard to correlate the performance degradation to FS when a program becomes very large, and even harder to identify the data structure and codes that cause the FS.

Figure 4.1 shows an OpenMP version of one of the Phoenix benchmarks[2] [48], the linear regression kernel. The *schedule(static,chunk_size)* clause on the loop directive causes the iterations of the loop to be distributed to threads in a round-robin fashion, where each thread gets *chunk_size* number of iterations of the outer loop at a time.

[1] data object or codes from which false sharing occurs

[2] Phoenix benchmarks implement MapReduce for shared memory systems.

```
#define N 9600
#define M 28887
#pragma omp parallel for private(i,j) schedule(static,1)
  for(j=0; j<N; j++)
     for (i = 0; i < M/num_threads; i++)
     {
      tid_args[j].SX  += tid_args[j].points[i].x;
      tid_args[j].SXX += tid_args[j].points[i].x *
         tid_args[j].points[i].x;
      tid_args[j].SY  += tid_args[j].points[i].y;
      tid_args[j].SYY += tid_args[j].points[i].y *
         tid_args[j].points[i].y;
      tid_args[j].SXY += tid_args[j].points[i].x *
         tid_args[j].points[i].y;
     }
```

Figure 4.1: OpenMP version of the linear regression kernel

Figure 4.2 shows the execution time of the linear regression kernel with different chunk size configurations. The kernel clearly exhibits FS impact on performance because of the small chunk size. Since the chunk size is 1, threads execute consecutive iterations of the outer loop. Therefore, each thread accesses consecutive element of the *tid_args* array which causes FS. However as we increase the chunk size value, the execution time gradually decreases because threads will be accessing non-neighbor elements of the array. By increasing the values of the chunk size from 1 to 30, we are reducing the FS overhead and improving the execution time of the kernel by up to 30%.

Several different approaches have been taken in the area of detecting FS in a program. One approach is via cache simulation and memory tracing [35, 38, 30, 26, 22]. In this approach, compiler instruments the binary code with tracing routines, and a

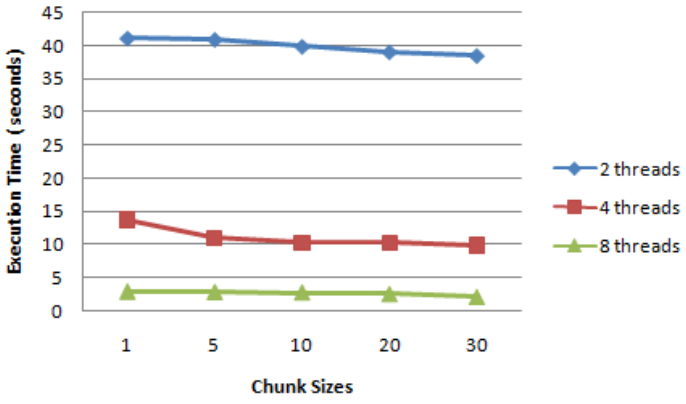


Figure 4.2: Execution time vs. chunk size of linear regression kernel.

tracing tool then captures the memory accesses performed at runtime and stores the tracing information for offline simulation. The tracing file is fed to a simulation tool which in turn simulates caches in the modeled architecture and determines the types and degree of cache misses that occurred during the simulation. However, tracing every memory reference performed during the execution of a program can degrade the overall execution performance greatly.

Analyzing performance of a program using hardware performance counters is another approach taken to detect false sharing in the code [59, 36]. Performance counters are used to detect the major bottleneck in the program and to identify reasons for the bottleneck. The main drawback of this approach is that the programmer has to understand the results obtained from the performance counters and manually identify the source of the problem in the code.

FS detection techniques implemented in [54] and [64] differ greatly from previous approaches. The technique in [54] write-protects the shared data; when a process attempts to write to the shared data, a page fault occurs, the tool copies the page and

the process accesses the local copy of the page. The tool detects FS by observing every word modified in local copies of pages that processes access. The approach in [64] uses a memory shadowing technique to track the cache access patterns.

There have been also research efforts to provide techniques for eliminating FS [23, 11]. In [11], the authors show how to mitigate FS by changing the scheduling parameters such as chunk size and chunk stride for parallel loops. In [23] several compiler transformations are described such as array padding and memory alignment that optimize the layout of data and decrease the possibility of FS to occur.

The goal of this work is to develop a compile-time cost model [29] for FS, and to estimate the performance impact of FS on parallel loops using the defined model. With the help of cost models, the compiler is able to estimate whether the specific transformation is profitable in terms of execution time and determine the optimal level of the transformation, if applied.

The FS cost model defined in this work features:

1. Ability to output the total number of FS cases that will occur during execution of the parallel loop

2. Ability to analyze the performance impact of FS on a parallel loop as a percentage of execution time

3. A linear regression model to reduce the modeling time by approximation without impacting its accuracy

4.2 Methodology

Our FS cost model estimates the number of FS cases in a parallel loop at compile-time, and computes the overhead cost incurred by the problem on the whole execution of the loop. For wide applicability, we use OpenMP parallel loops in this model. Given an OpenMP parallel loop, there are four steps to analyze the cost incurred by FS:

1. Obtain array references made in the innermost loop of a loop nest
2. Generate a cache line ownership list for each thread
3. Apply a stack distance analysis to each cache line ownership list
4. Detect false sharing

As we mentioned, FS is often only revealed at runtime and is sensitive to lots of details about how the program is being executed, e.g. the alignment of allocated memory, the number of threads working on the victim data, and other background applications that may compete for the cache resources. It is necessary to supply enough runtime information to the compiler when estimating the FS effects. In this model, the compiler needs information about the number of threads executing the loop, loop boundaries, step sizes, index variables, and the *chunk size*, if specified, for the OpenMP parallel loop. The chunk size is the number of iterations of a loop that are distributed to each thread. We assume that chunks of a loop are distributed to threads in a round-robin fashion. If the loop boundaries are not known at compile-time, the model only outputs the FS rate estimated per full cycle

of iterations executed by all of the threads. One full cycle of iterations executed by the thread team is the sum of iterations executed by each thread in one chunk size.

4.2.1 Obtain Array References from the Source Code

Our FS model identifies FS caused by only array references made in the innermost loop. The details about each array reference such as array base name, indices, type of access (read/write) are collected by a compiler and stored in an array reference list. However, more specific information such as which thread will access which region/elements of an array will be generated automatically in the next step of the model.

4.2.2 Generate a Cache Line Ownership List

At each iteration of a chunk of a loop a cache line ownership list is generated using the array reference list and the new values of loop indices. The cache line ownership list contains information about which cache line is being read/written by a thread at that specific iteration. The assumption we make at this step is that all array variables are aligned with the cache line boundary, so that it would be possible to know the relative cache lines on which array elements are located at compile-time.

4.2.3 Apply a Stack Distance Analysis

We create a separate cache state for each thread at the beginning of our FS modeling technique. The cache states store the current state of private caches that threads operate upon. When a new cache line ownership list is generated for each thread (in the previous step), we update the cache states with new cache line ownership lists and apply a stack distance analysis on cache states. The stack distance [51] is the number of distinct cache lines accessed between two accesses to the same cache line. The distance of the stack is the total number of cache lines for a fully associative cache or number of lines in one set for a set associative cache. Basically, the stack distance analysis simulates the least recently used (LRU) cache and outputs the state of the cache at each distinct point of time. Before one element of the cache line ownership list is inserted into the cache state, the analysis checks whether the cache line already exists in the cache state or not. If so, the analysis moves the position of the existing cache line to the top of the stack; otherwise, it simply inserts the cache line at the top of the stack. If the number of distinct cache lines exceeds the stack size, the analysis evicts the cache line placed on the bottom of the stack, which is the LRU cache line. For FS model, the stack distance analysis simulates the fully associative cache. We simulate fully associative caches because modeling the fully associative cache is mostly valid especially for caches with a high level of associativity [50].

When the cache line is inserted into the cache state, our FS cost model proceeds to the next step which is to determine the number of FS cases that would occur at that specific iteration.

4.2.4 Detect False Sharing

Our model determines whether FS happens by performing a *1-to-All* comparison between the newly inserted cache line and the cache lines that other cache states already contain. For that purpose assume the following:

$$\begin{aligned} & cs_k \in S \\ & cl_i \in CLOL_k, k = 0, 1, 2, .., num\_of\_threads \end{aligned} \tag{4.1}$$

where cs_k is a cache state of thread k; S is a set of all cache states; $CLOL_k$ is a cache line ownership list of thread k; and cl_i is a cache line element of $CLOL_k$. We define a function $\varphi(cs_k, cl_i)$ as

$$\varphi(cs_k, cl_i) = \begin{cases} 1, if(cl_i \in cs_k \ \ and \ \ cs_k^{cl_i} = W) \\ 0, otherwise \end{cases} \tag{4.2}$$

which returns 1 if a cache state cs_k includes the cache line cl_i and the state of the cache line in the cache state - $cs_k^{cl_i}$ is modified; otherwise the function returns 0. Using the function $\varphi(cs_k, cl_i)$ our model determines whether FS happened or not. In order to compute the number of FS cases that would occur during one iteration, the model needs to execute the function until all cache lines in all threads' $CLOL_k$ lists are evaluated. Thus the number of FS cases occurring at one iteration can be determined using Equation 4.3.

$$false\_sharing_{iter} = \sum_{j=0}^{k-1} \sum_{i=0}^{n} \varphi(cs_j, cl_i) \times mask(cs_j, cl_i)$$
$$mask(cs_j, cl_i) = \begin{cases} 0, if(\mathrm{cl}_i \in \mathrm{CLOL}_j^{cs_j}) \\ 1, \text{otherwise} \end{cases} \tag{4.3}$$

The $mask(cs_j, cl_i)$ function ensures that the newly inserted cache line is not compared with the cache lines of the same cache state.

In order to estimate the total number of FS cases that would occur throughout the whole loop, our model needs to evaluate $\frac{All\_num\_of\_iters}{num\_of\_threads}$ number of iterations where for each iteration it needs to perform the steps 2-4 and store the FS cases estimated at that iteration.

4.2.5 Prediction of False Sharing using Linear Regression Model

When the number of iterations of a loop is large, our FS model might take quite a long time to estimate the total number of FS cases. This is because the model needs to evaluate $\frac{All\_num\_of\_iters}{num\_of\_threads}$ number of iterations.

To overcome this limitation, we propose a FS prediction model that predicts the total number of FS cases by evaluating fewer iterations in much less time. The prediction model uses a linear regression model [18]. Figure 4.3 shows that the estimated number of FS cases increases linearly when different *chunk runs* are evaluated, where a single chunk run refers to $chunk_{size} \times num\_of\_threads$ iterations. Since the relation

between chunk runs and the estimated FS cases is linear, the linear regression model is suitable for use in our cost model.

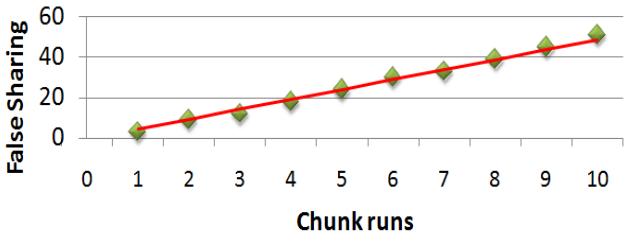


Figure 4.3: The linear increase of false sharing cases with the number of chunk runs

The prediction of the total number of FS cases using the linear regression model is as follows:

- We denote n iterations already evaluated by the model as $\vec{x} = \{x_1, ..x_n\}$; and estimated FS cases in n iterations as $\vec{y} = \{y_1, ..y_n\}$.
- The FS prediction can be modeled via $\vec{y} = a\vec{x} + b$ where initial $\vec{y}$ and $\vec{x}$ are known.
- We want our predicted results to be very close to the estimated results i.e. the error between predicted and estimated be minimal. The *Least Square Solution* therefore suggests the function to be $f = \left\|a\vec{x} + b - \vec{y}\right\|_2 = (a\vec{x} + b - \vec{y})^T(a\vec{x} + b - \vec{y})$.
- We have to find a and b such that the result of the function f is minimal which is $a, b = \arg\min_{a,b} f(a, b) = (a\vec{x} + b - \vec{y})^T(a\vec{x} + b - \vec{y})$.
- Thus we differentiate the function f with respect to a and b and obtain $a = \sum_{i=0}^{n-1} x_i y_i / \sum_{i=0}^{n-1} (x_i)^2$, $b = \sum_{i=0}^{n-1} y_i - \frac{a}{n} \sum_{i=0}^{n-1} x_i$.

- After we compute a and b, we can predict $y_{\max}$, which is the total number of FS cases, using $y_{\max} = ax_{\max} + b$ where $x_{\max}$ is represented either as the maximum number of iterations or chunk runs of a loop.

4.3 Evaluation and Results

The false sharing cost model was implemented within the Open64 compiler's LNO phase. We implemented a separate compiler pass that is applied to the high level IR to collect information needed to perform the modeling, including the information about the parallel OpenMP loop nest as well as memory loads/stores performed in the body of the innermost loop. The details about the loop nest include loop boundaries, step sizes, loop index variables and the chunk size, if specified, for OpenMP loops. Memory load/store information is collected by traversing the IR and obtaining array reference details such as array base name, array index, and memory offsets for arrays storing structured data types. The analysis we implemented did not require any modification to the compiler's IR, and the information about loop nest and memory operations are generally available in most compiler IRs. Thus we believe that our cost model can be implemented in other compilers with a similar approach as well.

4.3.1 Methodology for Accuracy and Efficiency Evaluation

Our FS cost model is evaluated in terms of both accuracy and efficiency. To demonstrate the accuracy, we compared the percentages of measured and modeled FS

overhead costs on the total loop execution time. We expect that the measured percentage of FS overhead should be close to the percentage of FS overhead modeled by our FS cost model as shown in Equation 4.4.

$$\frac{T_{fs\_measure} - T_{nfs\_measure}}{T_{fs\_measure}} \approx \frac{N_{fs\_model} - N_{nfs\_model}}{N^*_{fs\_model}} \tag{4.4}$$

where $T_{fs\_measure}$ is the measured time needed to execute a loop exhibiting FS; $T_{nfs\_measure}$ is the measured time needed to execute the same, but optimized, loop that does not incur any FS; N_{fs_model} is the number of FS cases estimated by our model on a loop exhibiting FS; N_{nfs_model} is the number of FS cases estimated by our model on an optimized loop; $N^*_{fs\_model}$ is a normalized value for FS cases estimated by our model on a loop incurring FS.

The measured FS overhead cost is obtained by executing a kernel loop with two different chunk sizes and calculating the percentage as follows:

$$\frac{T_{fs\_measure} - T_{nfs\_measure}}{T_{fs\_measure}}$$

Two different chunk sizes represent a loop with FS and non-FS cases. For example, for one kernel in our experiments we are using chunk sizes 1 and 64, where a loop with chunk size=1 represents a loop with FS case, and a loop with chunk size=64 refers to a loop with non-FS case. A loop with FS case heavily suffers from FS, whereas a loop with non-FS case incurs less (or no) false sharing because we believe

that increasing the chunk size decreases the FS. The execution times of the loop with FS and non-FS cases are represented by $T_{fs\_measure}$ and $T_{nfs\_measure}$, respectively.

The modeled FS overhead cost is obtained by our FS cost model, which evaluates $\frac{All\_num\_of\_iters}{num\_of\_threads}$ iterations of a loop to compute the total number of FS cases for both FS and non-FS case loops. The computed percentage value is then calculated as follows:

$$\frac{N_{fs\_model} - N_{nfs\_model}}{N^*_{fs\_model}}$$

To demonstrate the efficiency of our FS prediction model, which uses the linear regression model, we compare the predicted amount of FS cases when a very small number of iterations are evaluated against the modeled total amount of FS cases when $\frac{All\_num\_of\_iters}{num\_of\_threads}$ iterations are evaluated. The smaller the difference between the predicted and the modeled values, the more efficient our prediction model is.

4.3.2 Experimental Results

Our experiments are conducted on a system with four 2.2 GHz 12-core processors (48 cores in total). Each core has dedicated L1 and L2 caches of 64KB and 512KB respectively; L3 cache of 10240KB size is shared among 12 cores. All the caches at the three levels have the same cache line size, 64 bytes, which satisfies the assumption in our cost model.

We have used OpenMP versions of *heat diffusion* [5], *discrete fourier transform (DFT)* [46] and *linear regression* [48] programs for our experiments. The

computation-intensive loop kernels of these programs, when parallelized with OpenMP, exhibit extensive data accesses to the boundary of each data segment that is processed by each OpenMP thread. These accesses could incur large occurence of FS during program execution if parameters such as the chunk size are not properly chosen when parallelizing the code.

Table 4.1: Comparison of % of false sharing overheads incurred in heat diffusion kernel

# of threads	Measured Time with chunk size=1 FS case (sec)	Measured Time with chunk size=64 non-FS case (sec)	Measured FS effect on execution time (%)	Modeled FS cases effect (%)
2	0.3593	0.2901	19.2%	6.9%
4	0.2263	0.1646	27.2%	6.9%
8	0.1639	0.156	4.8%	6.9%
16	0.6586	0.6205	5.7%	7.0%
24	1.0049	0.9564	4.8%	7.1%
32	1.4671	1.3608	7.2%	7.2%
40	1.8455	1.6130	12.5%	7.2%
48	2.247	2.1501	4.3%	7.2%

Tables 4.1, 4.2 and 4.3 show the results of our model compared to the measured FS effect. The measured execution times of the heat diffusion kernel with FS and non-FS cases are depicted in the second and third columns of Table 4.1, respectively. Using the execution time information, the measured FS effect (on the total execution time of the kernel) is calculated and shown in the fourth column of the table. The last column, on the other hand, shows the FS effect modeled by our cost model. The accuracy of the model is assessed by comparing the fourth and fifth columns against each other. The closer the values in both columns are, the more accurate our cost model is.

Table 4.2: Comparison of % of false sharing overheads incurred in DFT kernel

# of threads	Measured Time with chunk size=1 FS case (sec)	Measured Time with chunk size=16 non-FS case (sec)	Measured FS effect on execution time (%)	Modeled FS cases effect (%)
2	2.0978	1.7624	15.9%	32.0%
4	1.762	0.9618	45.4%	31.6%
8	0.8976	0.6033	32.7%	31.5%
16	0.599	0.3688	38.4%	33.2%
24	0.5041	0.3163	37.2%	32.8%
32	0.4727	0.2827	40.1%	35.6%
40	0.4792	0.2669	44.3%	36.7%
48	0.4664	0.279	40.1%	35.8%

An important point worth mentioning here is that loop kernels in heat diffusion and DFT programs are parallelized at the innermost loop level, while the loop kernel in linear regression program is parallelized at the outermost loop level. Results for heat diffusion and DFT kernels given in Tables 4.1 and 4.2, show that the modeled FS overhead percentages estimated by our cost model are close to the measured FS overheads, indicating that by modeling FS, we can accurately estimate the FS overhead cost at compile-time. However, due to the difference in loop parallelization style, we see that the modeled and the measured FS overhead percentage results for the linear regression kernel depicted in Table 4.3 are not close to each other. Moreover, one can observe that the modeled FS cases effect decreases proportionally with increasing number of threads. This is due to the fact that the total number of chunk runs that threads will execute in linear regression kernel is

$$x_{\max} = m/(num\_of\_threads * chunk\_size)$$

Table 4.3: Comparison of % of false sharing overheads incurred in linear regression kernel

# of threads	Measured Time with chunk size=1 FS case (sec)	Measured Time with chunk size=10 non-FS case (sec)	Measured FS effect on execution time (%)	Modeled FS cases effect (%)
2	0.4302	0.4135	3.8%	16.1%
4	0.118	0.1074	9.0%	14.7%
8	0.0421	0.0331	21.2%	9.0%
16	0.02	0.0182	8.8%	4.9%
24	0.0116	0.01	13.9%	3.3%
32	0.0079	0.0068	13.4%	2.5%
40	0.0062	0.0051	18.0%	2.0%
48	0.0055	0.0046	15.6%	1.7%

whereas in heat diffusion and DFT is

$$x_{\max} = (m * n)/(num\_of\_threads * chunk\_size)$$

where m and n are the upper bounds of the outer and inner loops, respectively. Thus, in the loop kernel of the linear regression program, the number of chunk runs and consequently the total number of FS cases are directly dependent on the number of threads.

Results in Tables 4.4, 4.5 and 4.6 show the comparison between FS effects obtained from both the FS model and the FS prediction model, which uses the linear regression model. When modeling the effects with predictions, the number of FS cases is estimated with fewer iterations. For example, when the heat diffusion kernel is executed with 8 threads, with chunk run=20 and chunk sizes 1 and 64, our prediction model evaluates 8*1*20 and 8*64*20 iterations, respectively. On the other hand, without the prediction model, our FS model would evaluate $\frac{All\_num\_of\_iters}{num\_of\_threads}$ iterations.

Table 4.4: Comparison of predicted vs. modeled false sharing cases and their overhead %'s in heat diffusion kernel

# of threads	Pred. # of FS cases chunk size=1 (chunk run=20)	Pred. # of FS cases chunk size=64 (chunk run=20)	Pred. FS cases effect	Modeled # of FS cases chunk size=1	Modeled # of FS cases chunk size=64	Modeled FS cases effect
2	91,991K	1,595K	6.8%	94,421K	2,107K	6.9%
4	92,979K	1,625K	6.8%	94,446K	2,145K	6.9%
8	93,496K	1,702K	6.8%	94,458K	2,070K	6.9%
16	93,990K	1,724K	6.9%	96,043K	1,888K	7.0%
24	94,155K	1,609K	6.9%	96,938K	1,699K	7.1%
32	93,986K	1,456K	6.9%	97,159K	1,509K	7.2%
40	94,286K	1,826K	6.9%	97,730K	1,889K	7.2%
48	94,319K	1,107K	7.0%	97,935K	1,126K	7.2%

Thus, if there were 5000*5000 iterations in total, our model would need to evaluate 3,125,000 iterations to compute the total number of FS cases. However, with the FS prediction model we would need to evaluate only 160 or 10,240 iterations, for chunk sizes 1 and 64 respectively. Our prediction model clearly facilitates the process of estimating the number of FS cases. The percentage results of predicted and modeled FS impacts depicted in Tables 4.4, 4.5, and 4.6 are very close to each other, indicating that our FS prediction model is accurate and efficient.

We have also included Figures 4.4 and 4.5 that show the false sharing effects (percentage of execution time) obtained through the execution measurement, the compile-time modeling, and the modeling using linear regression predictions. These results demonstrate the effectiveness of the model.

Table 4.5: Comparison of predicted vs. modeled false sharing cases and their overhead %'s in DFT kernel

# of threads	Pred. # of FS cases chunk size=1 (chunk run=50)	Pred. # of FS cases chunk size=16 (chunk run=50)	Pred. FS cases effect	Modeled # of FS cases chunk size=1	Modeled # of FS cases chunk size=16	Modeled FS cases effect
2	52,233K	26,468K	32.4%	53,058K	27,358K	32.0%
4	52,697K	26,491K	32.8%	53,088K	27,702K	31.6%
8	52,928K	26,612K	32.8%	53,311K	27,882K	31.5%
16	52,936K	26,526K	32.9%	54,411K	27,257K	33.2%
24	52,967K	27,475K	31.8%	54,956K	28,003K	32.8%
32	52,983K	25,523K	34.2%	55,245K	25,865K	35.6%
40	53,077K	24,895K	35.1%	55,510K	25,154K	36.7%
48	52,998K	25,649K	34.1%	55,542K	25,878K	35.8%

To summarize, we validated our model by comparing the FS overhead percentages obtained by measuring from the execution time against the ones computed by our model. The modeling results are comparable to the real execution behavior from 2 to 48 threads tested, showing the model can accurately quantify the FS impact at compile-time. The FS cost model will be used by compilers to guide the parallel loop transformations by providing more accurate timing estimation for parallel loops. Compilers will also be able to use information from our model to perform automatic optimizations, such as changing the loop iteration behavior or data alignment to eliminate FS or minimize its impacts. These modeling and estimation results could also be useful for programmers for performance tuning and locality optimizations. The quantitative performance impact information will be especially helpful when tuning an application for specific hardware architectures.

Table 4.6: Comparison of predicted vs. modeled false sharing cases and their overhead %'s in linear regression kernel

# of threads	Pred. # of FS cases chunk size=1 (chunk run=10)	Pred. # of FS cases chunk size=10 (chunk run=10)	Pred. FS cases effect	Modeled # of FS cases chunk size=1	Modeled # of FS cases chunk size=10	Modeled FS cases effect
2	85,592K	703	16.0%	86,315K	719	16.1%
4	77,561K	720	14.7%	77,685K	678	14.7%
8	47,473K	840	9.5%	44,545K	791	9.0%
16	24,804K	900	5.2%	23,274K	855	4.9%
24	16,778K	920	3.6%	15,771K	874	3.3%
32	12,667K	930	2.7%	11,907K	899	2.5%
40	10,172K	936	2.2%	9,579K	897	2.0%
48	8,497K	940	1.8%	7,987K	893	1.7%

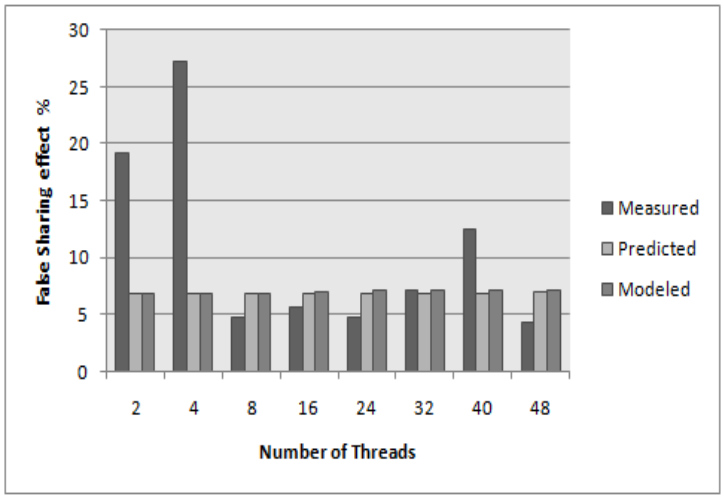


Figure 4.4: Comparison of %'s false sharing effects vs. different number of threads for heat diffusion kernel.

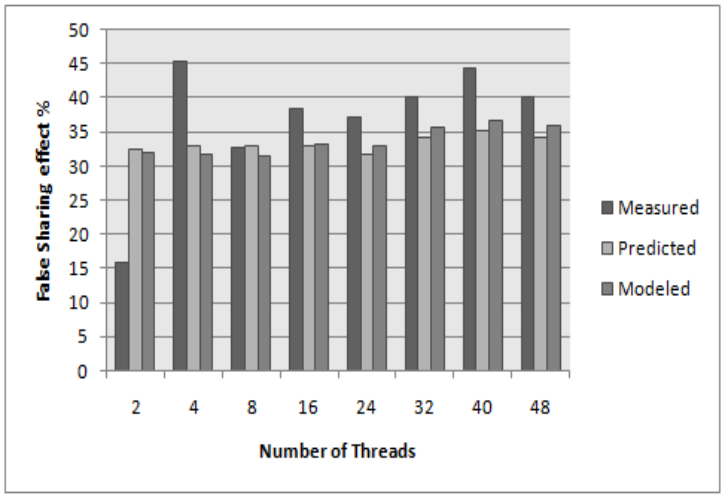


Figure 4.5: Comparison of %'s false sharing effects vs. different number of threads for DFT kernel.

Chapter 5

Off-Chip Memory Bandwidth Modeling

In this section, we discuss the importance of effectively using shared resources on an underlying architecture, specifically focusing on shared memory bus. We then introduce our approach of modeling the memory bandwidth utilization by parallel OpenMP applications at compile time. Finally, we discuss the experimental setup and the results obtained from evaluation of the model.

5.1 Introduction

Multicore and multiprocessor systems are designed to allow clusters of cores to share various hardware resources such as caches, memory bandwidth, and interconnects.

Efficient use of these resources requires both to maximize the sharing of these resources among concurrent threads, and also to minimize the contentions and conflicts of using them. Software tools including compiler and runtime systems play a very important role in optimizing and scheduling an application with regards to the resource sharing and usage conflicts.

There has been a large amount of previous work that focus on increasing the sharing of resources by runtime co-scheduling with the help of compiler optimizations [15, 16, 23, 40]. However, with the increase of the number of cores in a system, and the increasing depth of memory hierarchy that comes with higher non-uniformity of memory access, the inter-core resource conflict and contention increase as well. Reducing the conflict use of shared resources becomes critical when dealing with parallel applications on multicore and parallel systems. It is very important that an application does not demand resources more than the architecture can supply. If this is the case, the application may unnecessarily stall due to unavailability (contention) of some resources.

Off-chip memory bus is a shared resource that is commonly used among different cores on the same processor. Increasing the number of cores for a parallel application may not necessarily increase the performance of the application if the application requires more data than the memory bus can transfer at a time. High contention for memory bandwidth may even cause a significant performance degradation in parallel applications [61, 34]. This situation is often referred to as memory bandwidth bottleneck [16].

In order to prevent any kind of bottleneck from happening, it is crucial to tune

the application behavior to fit the architectural characteristics. On multicore system, it is important for threads to not interfere with each other, but instead to collaborate and orchestrate with each other. Code optimizations should not optimize for single-threaded performance on multicore architecture because it will cause bottlenecks on the system. Instead, optimizations should focus on improving multi-threaded performance by balancing the utilization of shared resources efficiently.

The topic of improving the efficiency of memory bandwidth has been studied quite extensively in the literature. Reducing the memory bandwidth of applications has been studied in [1], [16], [37]. Compiler optimizations/techniques such as loop fusion, store elimination, storage reduction are one of the methods that are referred in those works for reducing the memory bandwidth pressure to the system. Techniques to improve overall system performance, such as optimal memory bandwidth partitioning techniques, have also been studied in [31], [42], [57].

Closely related work on modeling the off-chip memory bandwidth analytically or statistically has been performed in [27], [33, 34], [61]. Authors of [34] have used pChase benchmark to perform experimental multi-socket, multicore memory bandwidth study, using which they developed an analytical memory bandwidth model. The model characterizes memory bandwidth performance at three levels which are bandwidth per core, socket and node levels. The authors compared the experimental results obtained from the pChase benchmark against the modeled results for several multi-socket, multicore architectures. Their goal was to model bandwidth performance for various architectures by generating a model using the pChase benchmark

and comparing the model against the pChase benchmarking results for different architectures. In our work we aim at modeling bandwidth performance for various loop kernels by generating a model based on the STREAM benchmark and predicting the bandwidth performance based on a loop signature (# of threads and # of concurrent cache misses).

Authors of [33] analyzed the effect of memory controllers on a local and remote memory bandwidth performance, and developed a model to evaluate the performance of on-chip and cross-chip interconnect of a multicore processor. Their model predicts the memory bandwidth performance based on the number of processes running on local and remote nodes. Authors showed that in some cases, accessing data via the cross-chip interconnect may be more advantageous than accessing it from the local node in terms of the bandwidth usage. Our work at this time does not consider cases when the data is accessed both locally and remotely. Nevertheless, we are planning to work on the idea in the near future.

Authors of [61] proposed a performance model for OpenMP, MPI, and hybrid applications based on the memory bandwidth contention and communication time. The model predicts the execution time of an application. The authors first measured the memory bandwidth for one and two cores using the STREAM benchmark. Using the performance data obtained from one and two cores, referred as baseline values, the model predicts the execution time of an application on higher number of cores. Their model, essentially, relies on the memory bandwidth ratio of higher cores to baseline values. However, we believe that the memory bandwidth ratio does not increase linearly as the number of active cores increases. Therefore, our model does

not rely on initial baseline values when predicting for higher number of cores.

In [27] an analytic model to estimate the optimal cache size and the memory bandwidth for many-core based systems is proposed. The model is used to estimate the lower-level memory bandwidth given the upper-level cache size and the statistical behavior of a program. The model uses central limit theorem and the stochastic behavior of cache misses that is generated using traces during simulation.

In this chapter, we present an off-chip memory bandwidth model developed to estimate the bandwidth requirement of a parallel loop at runtime. This mechanism uses statistical polynomial curve fitting technique on a set of bandwidth measuring data to derive the model that can be applied to other applications. The main support for our model is provided by a compiler analysis to estimate the number of memory accesses that would result in a cache miss.

We make the following contributions:

- We introduce a modified STREAM kernel that is used with the curve fitting technique to derive the statistical memory bandwidth model for a particular system with regards to the parallelism and concurrent cache misses.
- We propose a compile-time statistical model that can be used to predict the memory bandwidth requirement of parallel loops when being executed with specific number of threads.

The model can serve as a cost model to determine an optimal configuration of concurrent memory accesses and the number of threads to run a memory-intensive

loop with in order to prevent the memory bandwidth bottleneck. Knowing maximum number of concurrent memory accesses per thread would help programmers, compilers as well as performance tuning tools to evaluate the benefits of certain level of compiler optimizations. Knowing the memory bandwidth performance with respect to the number of threads would also help in deciding the best configuration of threads to run the application with.

5.2 Methodology

5.2.1 Memory Bandwidth Analysis

Computer vendors often provide a theoretical (peak) bandwidth of the memory system of a machine, and practically, the sustainable bandwidth is used to represent performance of a memory system. The sustainable bandwidth could be determined by performing benchmarking experiments with varied number of threads and number of memory accesses, for example, using the Triad kernel from STREAM benchmark [39] shown in Figure 5.1. The sustainable memory bandwidth is computed using the Equation 5.1 where $T_2 - T_1$ is the time it takes to run the kernel.

$$Bandwidth = \frac{Data_{transferred}}{T_2 - T_1} \tag{5.1}$$

The sustainable memory bandwidth represents the maximum bandwidth of the memory system that is available to applications. However, an application often exhibits different memory bandwidths during its execution within the sustainable

```
double a[N]; double b[N]; double c[N];
T1 = tick()
#pragma omp parallel for
for (i=0; i<N; i++) {
    a[i] = b[i] + q*c[i];
}
T2 = tock()
```

Figure 5.1: Original STREAM Triad kernel

bandwidth. To create a model for measuring the application memory bandwidth, we slightly modified the STREAM Triad kernel to allow us to control the number of memory accesses, as shown in Figure 5.2. Let us assume that the default cache line size is 64 bytes. Figure 5.2 shows that three array references of each iteration of the parallel loop will result in three cache misses, referred to as *concurrent cache misses* of each thread. Then in the Equation 5.1, the total amount of data transferred would be equal to *N * # of concurrent misses * cache line size*.

```
double a[N][100]; double b[N][100]; double c[N][100];
T1 = tick()
#pragma omp parallel for
for (i=0;i<N;i++) {
  a[i][0] = b[i][0] + q*c[i][0];
}
T2 = tock()
```

Figure 5.2: Our version of STREAM Triad kernel

Unlike the original Triad kernel, using the *concurrent cache misses* approach, we are able to co-relate the required memory bandwidth to the number of cache misses happened in one iteration of the loop by each thread. This is an important parameter

that reflects the data access pattern of arrays in the loop. Programmers can change the access pattern of the loop body, thus to exhibit different memory bandwidth requested. In our example, in order to observe different memory bandwidth requirements, one needs to increase the number of concurrent cache misses per iteration by duplicating the statement in the loop block and changing the array indices as in Figure 5.3.

```
double a[N][100]; double b[N][100]; double c[N][100];
#pragma omp parallel for
for (i=0;i<N;i++) {
  a[i][0] = b[i][0] + q*c[i][0];
  a[i][16] = b[i][16] + q*c[i][16];
  a[i][32] = b[i][32] + q*c[i][32];
  ...
}
```

Figure 5.3: The STREAM Triad kernel with increased number of concurrent cache misses

By duplicating the statements in the loop block as in Figure 5.3, we increase the number of *concurrent cache misses* to be 9 for this case, since all of the array references made in the loop block will result in a cache miss. In this way, we are subsequently increasing the memory bandwidth requirement per iteration.

Using the modified Triad kernel in Figure 5.3, we measured the memory bandwidth performance with regards to the number of threads and the number of *concurrent cache misses*, and also studied the effect of data and thread placement on the overall bandwidth performance. All experiments were performed on a Non-Uniform

Memory Access (NUMA) system whose architecture is depicted in Figure 5.4. The system has two 2.2GHz 12-core AMD Opteron processors, where each processor has two 6-core CPU chips. The four CPU chips are inter-connected through cache-coherent HyperTransport (HT) links, HT_1, HT_2 and HT_3. The three HT links differ in size such as $HT_3 = \frac{HT_2}{2} = \frac{HT_1}{3}$ where HT_3, HT_2 and HT_1 links connect *node0* to *node3*, *node0* to *node2* and *node0* to *node1*, respectively.

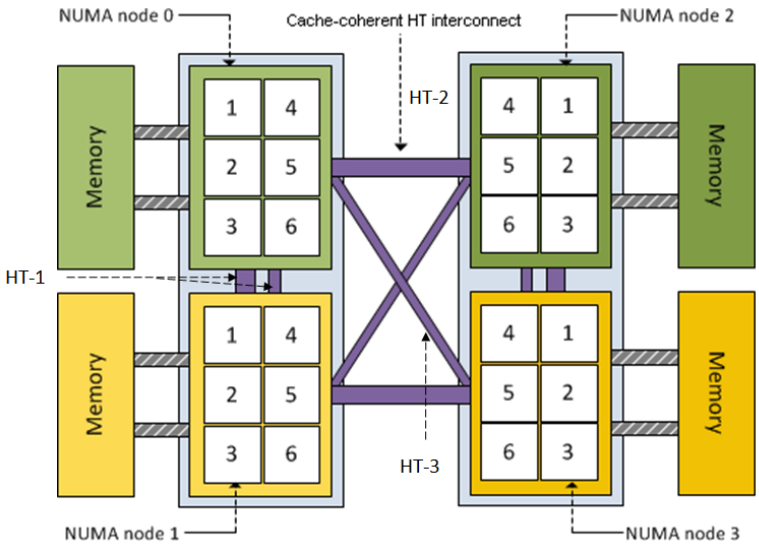


Figure 5.4: Memory bandwidth evaluation platform

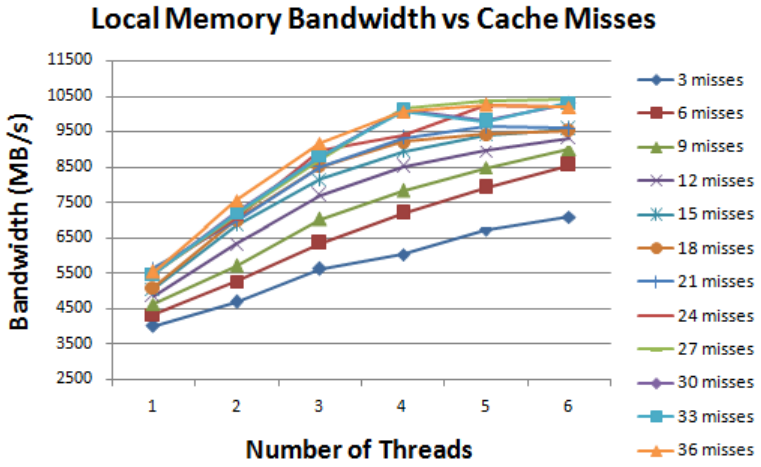
Figure 5.5a shows the memory bandwidth measured on a local memory channel when threads access the data located on the same NUMA node. This is achieved by executing the program via *numactl –cpunodebind=0 –membind=0 ./executable* command where we deliberately specify both the data and the threads to run on the same node. Figures 5.5b, 5.6a, 5.6b show the bandwidth performance achieved via each of the three HT links, i.e. when threads access the data located on another

node of the NUMA system. The bandwidth performance results in these figures are represented with respect to the number of threads executed the loop and the number of concurrent cache misses occurred per iteration of the loop.

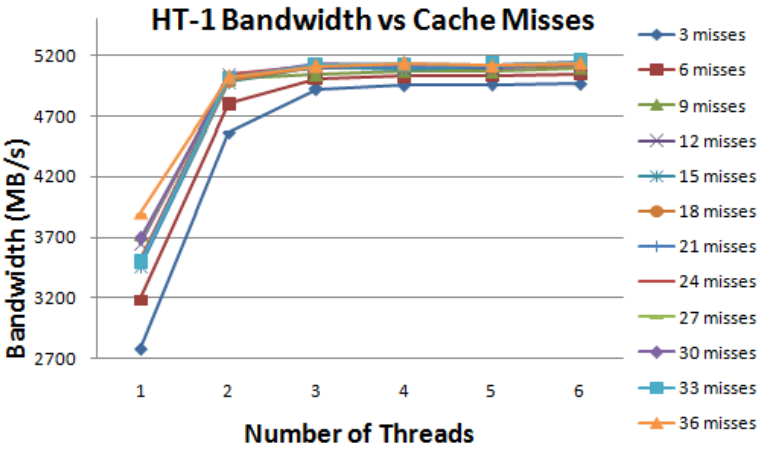
These experimental results showed that, in general, the maximum memory bandwidth can be reached quite easily with a small number of threads and a small number of concurrent cache misses. When this happens, increasing the number of threads and/or the number of concurrent cache misses won't necessarily increase the memory performance, but perhaps degrade the performance due to the resource contention. Therefore, knowing when the sustainable memory bandwidth is reached when executing a parallel loop under specific configurations (number of threads and memory accesses in our example) would be helpful for programmers, compilers, and performance tuning tools to determine an optimal configuration of execution, thus improving the resource utilizations.

5.2.2 Memory Bandwidth Model

The experimental data obtained from benchmarking our modified Triad kernel allows us to correlate the parallelism (number of threads) and the data access patterns (concurrent cache misses) to the memory bandwidth required at a particular point of execution of an application. Using those data, we are able to derive a memory bandwidth model that accurately represents such correlation. Our proposed bandwidth model is based on a *polynomial curve fitting technique* [12]. In this approach, the results of our bandwidth experiments are considered as a collection of data that can

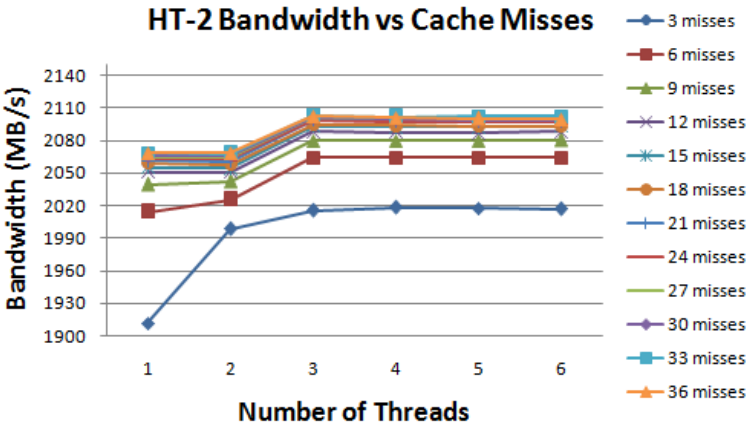


(a)

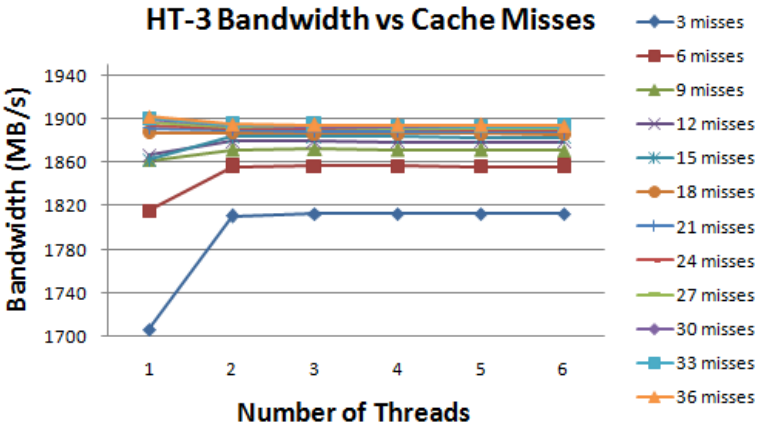


(b)

Figure 5.5: Memory bandwidth measured using modified Triad kernel through (a) local memory and (b) HT-1 links for different number of threads and cache misses.



(a)



(b)

Figure 5.6: Memory bandwidth measured using modified Triad kernel through (a) HT-2 and (b) HT-3 links for different number of threads and cache misses.

be represented as a function or a curve of interest. Using the curve fitting technique, we generate the best fit function of bandwidth to the input experimental data. Let τ represent the number of threads executing the loop and μ represent the number of concurrent cache misses occurring per iteration of the loop. Using the curve fitting technique, we determine the polynomial coefficients P of the curve.

$$f_\tau(\mu) = p_1\mu^n + p_2\mu^{n-1} + ... + p_n\mu + p_{n+1} \tag{5.2}$$

Given $P = \{p_1, p_2...p_{n+1}\}$ where n refers to the polynomial degree, in our case $n = 5$, our model can predict the required memory bandwidth f_τ for given number of threads and the number of concurrent cache misses using Equation 5.2.

For a given computer system, by performing the experimental analysis using our modified Triad kernel in Figure 5.3 and collecting memory bandwidth measurements, then applying the curve fitting technique, we can obtain a distinct function that becomes a memory bandwidth model of the system. Using the generated model in a compiler or performance estimating tool, we can predict the maximum sustainable bandwidth that an OpenMP loop can achieve for the given system.

We made the following assumptions in our model:

- Memory access dominates the execution time of the loop body, thus the time spent to perform other instructions are neglected in this model. Since memory access latencies are normally in hundreds of CPU cycles, this assumption is valid for scientific kernels that exhibit up to moderate computation intensity.

- When relying on compiler analysis to obtain the concurrent cache misses for a given loop, we assume the cache line size is 64 bytes and array variables used inside the loop are declared cache aligned.

- The cache is fully associative. Set associative caches are complicated due to the reason that knowing the corresponding line in a set an array reference will be placed at compile time is very challenging. Moreover, assuming the fully associative caches is mostly valid for high level associative caches [50].

We have also observed that the memory bandwidth does not depend on the number of iterations of the loop due to the Equation in 5.3.

$$\begin{aligned} Bandwidth &= data/time \\ &= (total\_iters * bytes\_per\_iter)/(time\_per\_iter * (total\_iter/\#threads)) \\ &= (\#\_threads * bytes\_per\_iter)/time\_per\_iter \end{aligned} \tag{5.3}$$

The Equation 5.3 in fact shows that the bandwidth depends solely on the number of threads, and the amount of data being transferred in one iteration as well as the time spent to execute one iteration of the loop. According to the equation, the sustainable bandwidth is not related to the number of iterations of a loop. If this assumption is correct, then using the same Triad kernel, which is a singly nested loop, we will be able to model memory bandwidth performance for other types of loops such as doubly and triply nested loops. This means that by obtaining experimental data using Triad kernel, we can use the same data to predict memory bandwidth for any

loop for a given system. We conducted a set of experiments to validate whether this assumption is satisfied in practice i.e. whether the memory bandwidth performance is really dependent on the number of iterations or not. The experimental results given in Table 5.1 show that our theoretical assumption is correct. This allows us to use the experimental data obtained from benchmarking Triad kernel in predictions of all other types of loops on the same system.

Table 5.1: Bandwidth versus I*J number of iterations

I (M=10^6, K=10^3)	J	Bandwidth (GB/s)
1.4 M	100	12.7
1.4 M	200	12.9
1.2 M	100	12.7
1.2 M	200	12.9
1 M	100	12.5
1M	200	12.9
800 K	100	12.9
800 K	200	12.9
600 K	100	12.6
600 K	200	12.7
400 K	100	12.9
400 K	200	12.7

The model predicts the bandwidth performance for one loop iteration i.e. the results are obtained for a specific iteration at a time. Accurately predicting an average bandwidth performance for all iterations combined is very challenging if there is large divergence of the behaviors of the loop body. In other words, a loop can exhibit high memory bandwidth requirement during some iterations, and require less bandwidth during other iterations. In this case, taking an average of these two

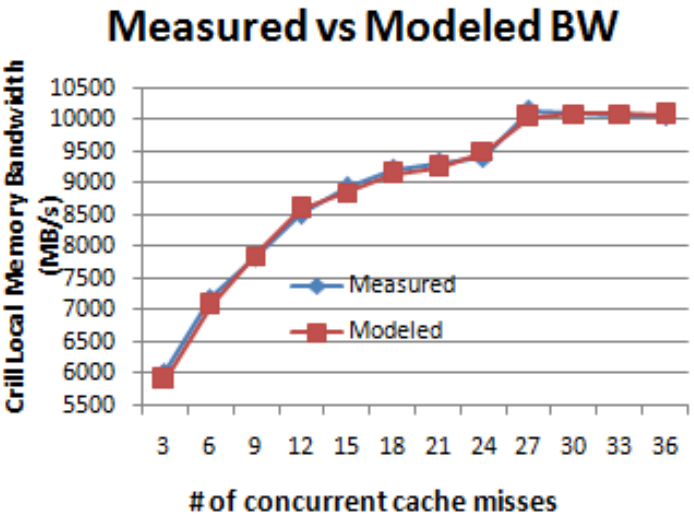
cases and outputting a single bandwidth performance for all the iterations would be inaccurate, because we would be missing the high memory bandwidth case. Instead, our model identifies the specific iterations where the bandwidth performance may change drastically, and outputs the predictions for these identified iterations only.

For a given OpenMP parallel loop, the concurrent cache misses are obtained via the same compiler analysis that was discussed in Section 4.2.

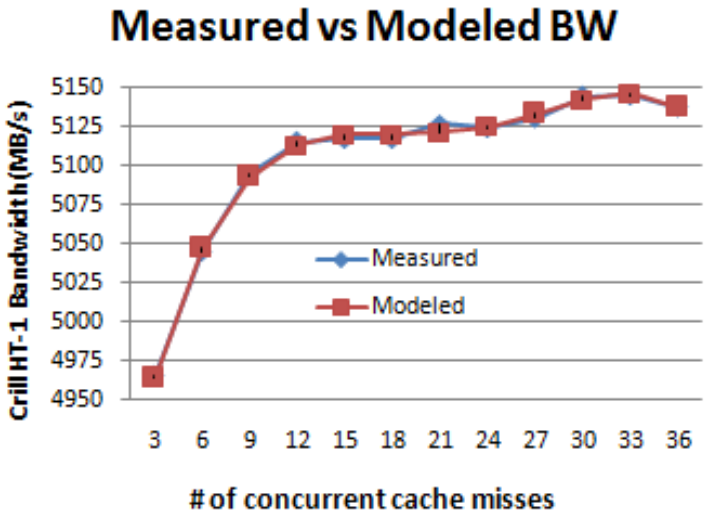
5.3 Evaluation and Results

Our model is evaluated in terms of both technical accuracy and prediction accuracy. To demonstrate the technical accuracy, we compared the measured and the modeled bandwidth results obtained from the Triad kernel experiments.

Figures 5.7 and 5.8 show the comparison of both the measured and the modeled bandwidth values for modified STREAM Triad kernel for different number of concurrent cache misses. The number of threads used in this experiment is 4. For this experiment, we first collected experimental bandwidth measurement data by running the modified Triad kernel with different numbers of concurrent cache misses. Then we generated a model using the curve fitting technique. We then obtained predicted bandwidth results for the same kernel using the generated model. In this way we assessed whether the curve fitting technique is appropriate and accurate enough to be used in our model. The closer the values between the measured and the modeled bandwidth results, the more suitable the technique is for our model. One can see that the curve fitting technique used in the model is very accurate, thus making the

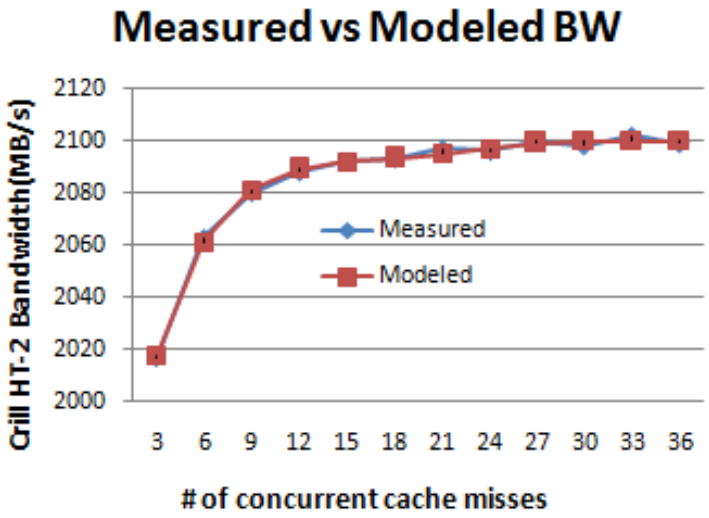


(a)

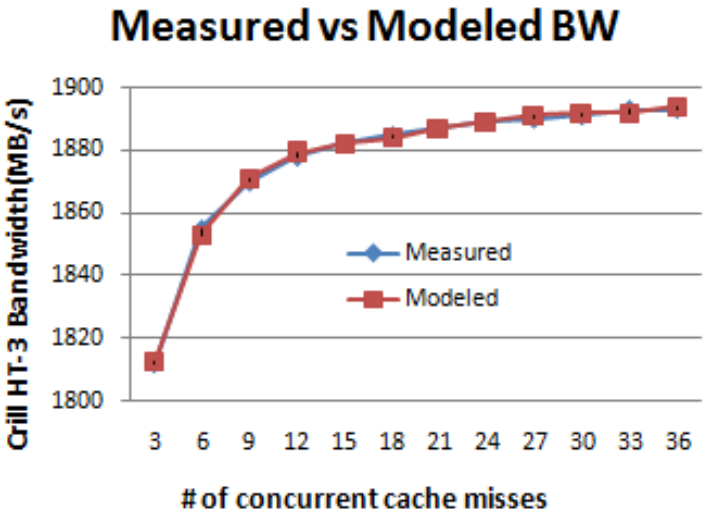


(b)

Figure 5.7: Comparison of measured and modeled bandwidths for modified STREAM Triad kernel via (a) local memory and (b) HT-1 links.



(a)



(b)

Figure 5.8: Comparison of measured and modeled bandwidths for modified STREAM Triad kernel via (a) HT-2 and (b) HT-3 links.

technical accuracy of the model very high.

To demonstrate the prediction accuracy, we evaluated our model using OpenMP parallel loops from several widely known applications including Jacobi method [49], Scalar Penta-diagonal solver (SP) and Multigrid kernel (MG) from NAS benchmarks [2]. We compared the measured bandwidth results obtained by running the applications against the bandwidth results estimated by our model. The modeled bandwidth results for the three applications are obtained using the same model that was generated from Triad kernel's bandwidth measurement data. Our model predicts bandwidth only for certain iterations of the loop. It is very challenging to accurately predict the bandwidth utilization for the whole execution time period of the loop at compile time. This is because the bandwidth utilization can change throughout the execution of the loop. Therefore, the model does not estimate the average bandwidth utilization; instead it estimates the memory bandwidth requirement for specific iterations at a time.

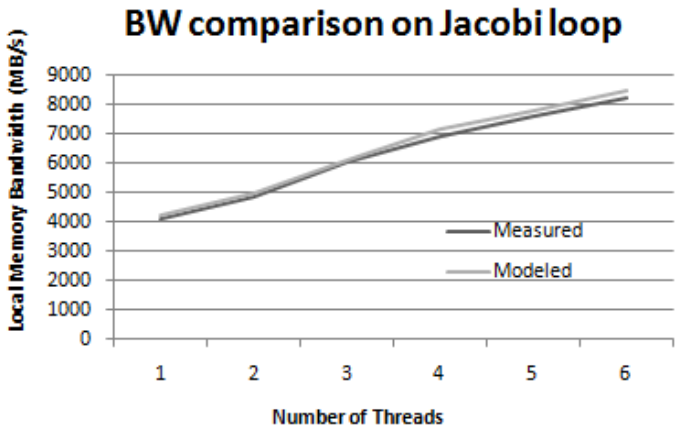
The Jacobi kernel is a doubly nested OpenMP loop, hence the number of cache misses per iteration varies based on the iteration being executed. Let (i,j) be a notation to represent an iteration where the outer and inner loops' indices are i and j, respectively. For the Jacobi loop, at the very first iteration (1,1), considered that a cache line size is 64 bytes and array elements are of size 8 bytes, four concurrent cache misses will occur. For period of (1,2) - (1,7) iterations, there will not be any cache misses. At (1,8) or any (1,m) iteration, where m is a multiple of 8, there will be four cache misses again. This pattern is continued till the loop reaches (2,1) or any (*,1) iteration. At each iteration our model re-analyzes the number of concurrent cache

misses based on the current cache states of each thread. As one can see, in the Jacobi kernel, not all iterations cause cache misses to happen, thus not all of the iterations require back and forth memory accesses. Therefore, it would be technically incorrect to estimate the memory bandwidth performance for all the iterations as a whole. Instead, our model estimates the bandwidth performance for specific iterations such as (1,1),(1,2),(1,8) etc. where the number of concurrent cache misses change, hence the bandwidth performance changes. Moreover, since (1,1) and (1,8) incur the same number of concurrent cache misses, we consider only one of them and eliminate the other. Iteration (1,2) does not incur any cache misses, thus the model does not perform any bandwidth predictions. In this way, our model estimates the bandwidth performance for (1,1) iteration only, which is depicted in Figures 5.9 and 5.10.

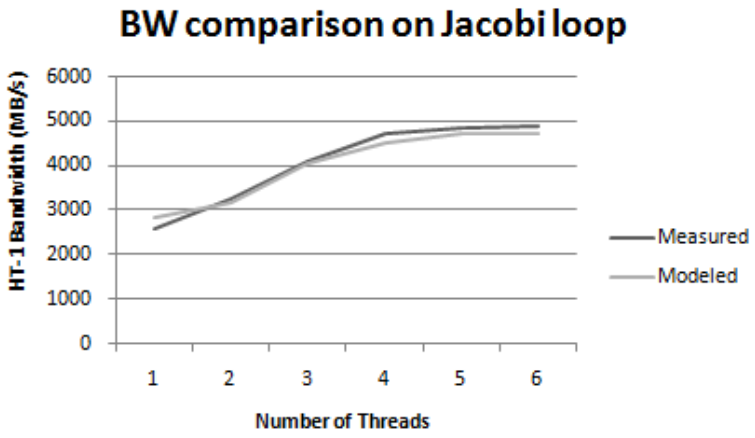
Figures 5.9 - 5.14 show the comparison between the modeled bandwidth estimations and the measured bandwidth values obtained from using different memory links for the Jacobi, MG, and SP kernels, respectively. The Jacobi kernel exhibits 4 cache misses at iterations (1,1), (1,8), (1,16), ..., (2,1), (2,8) and so on (as discussed earlier), so the modeled bandwidth performance results in Figures 5.9 and 5.10 are obtained for 4 concurrent cache misses. Figures 5.13 and 5.14, and 5.11 and 5.12, show the modeled bandwidth performance results obtained for 7 and 2 concurrent cache misses, respectively.

As a summary, in this chapter we presented our compile time off-chip memory bandwidth model and discussed how the defined model can be used to estimate the bandwidth performance of OpenMP parallel loops. We used the statistical polynomial curve fitting technique on a set of bandwidth measuring data obtained through

experiments by the modified STREAM benchmark. This model could be used by the compiler and performance tools to predict when the sustainable memory bandwidth of the system will be reached by the application during execution, and to determine an optimal number of threads that should be configured to execute a specific parallel loop according to its memory reference pattern. To evaluate our memory bandwidth model, we compared the measured and the modeled bandwidth performance results for several commonly used OpenMP kernels such as Jacobi, MG, and SP from NAS benchmarks. Our experimental results show that this model can be used for accurately estimating the memory bandwidth performance at compile time.

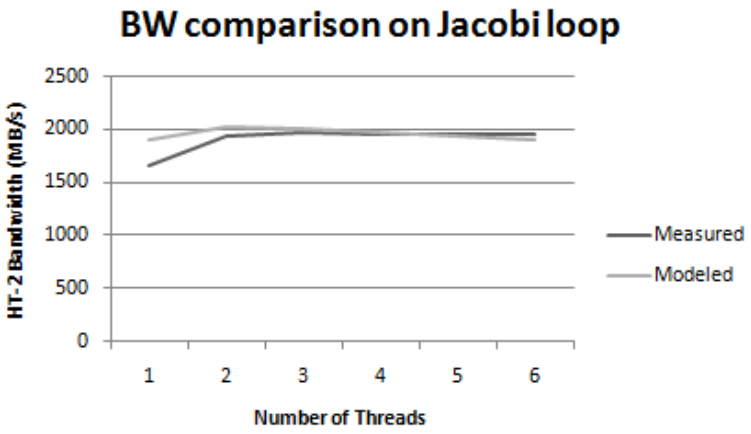


(a)

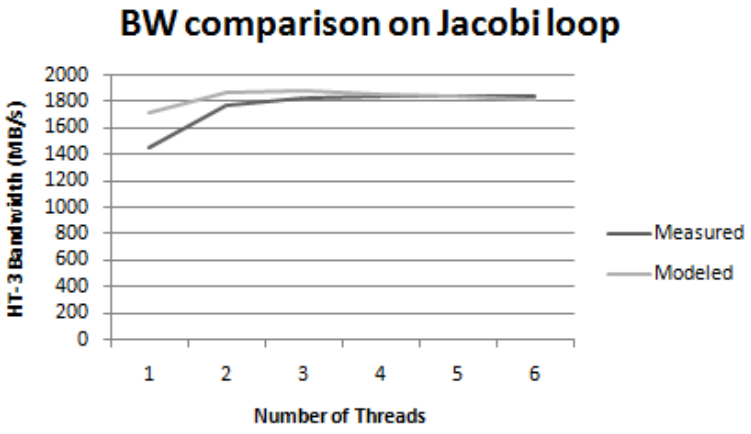


(b)

Figure 5.9: Comparison of measured and modeled bandwidths for Jacobi kernel via (a) local memory and (b) HT-1 links.

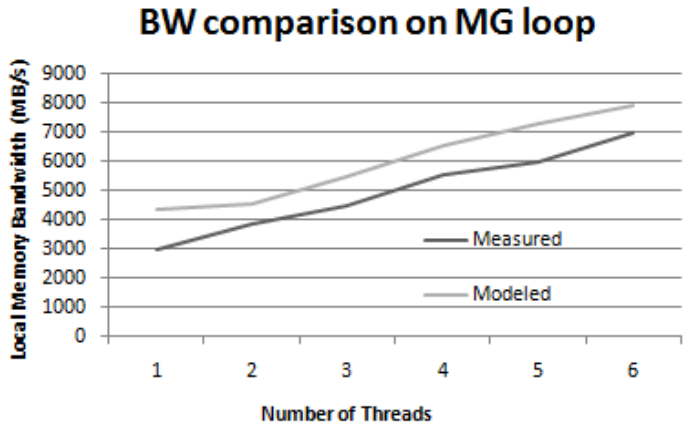


(a)

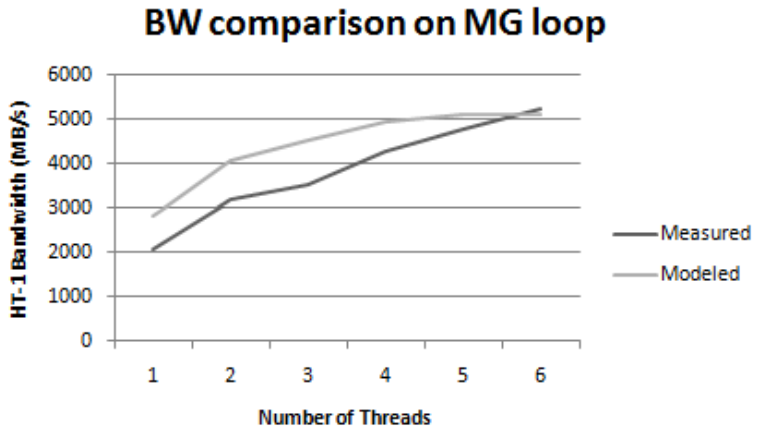


(b)

Figure 5.10: Comparison of measured and modeled bandwidths for Jacobi kernel via (a) HT-2 and (b) HT-3 links.

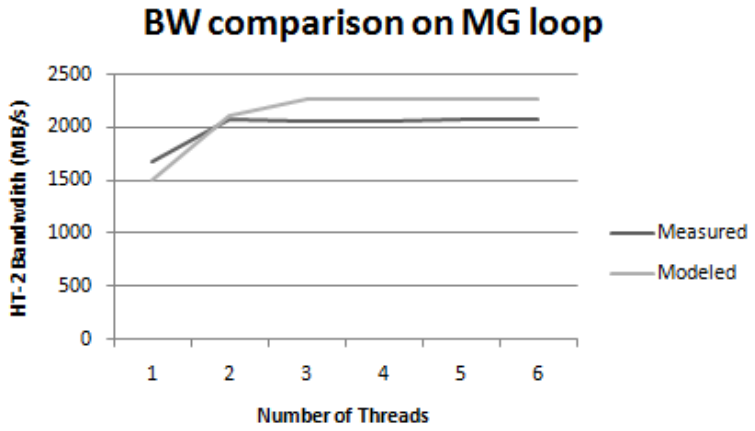


(a)

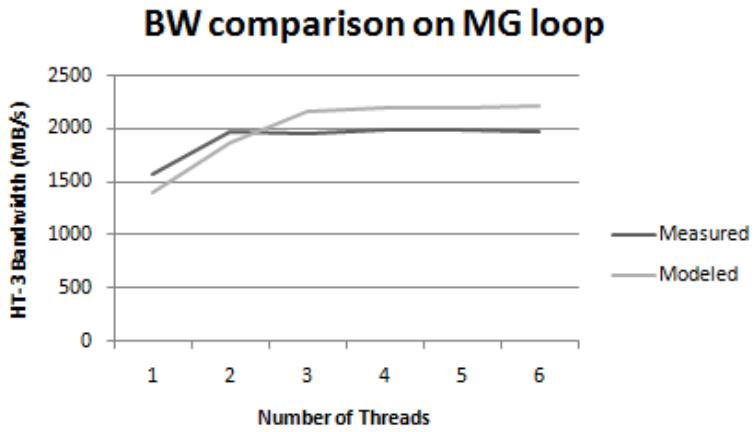


(b)

Figure 5.11: Comparison of measured and modeled bandwidths for a kernel from MG benchmark via (a) local memory and (b) HT-1 links.

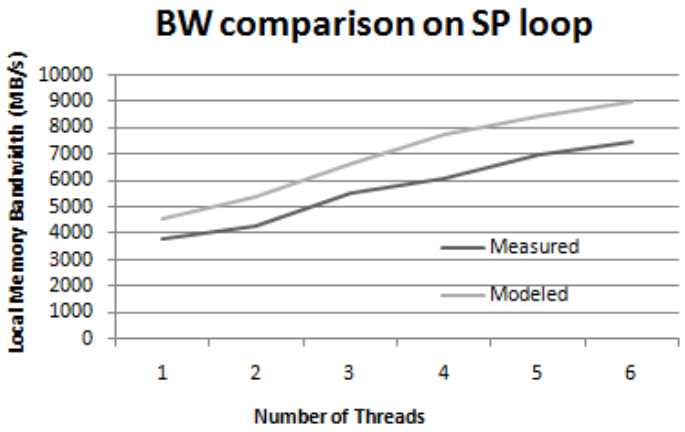


(a)

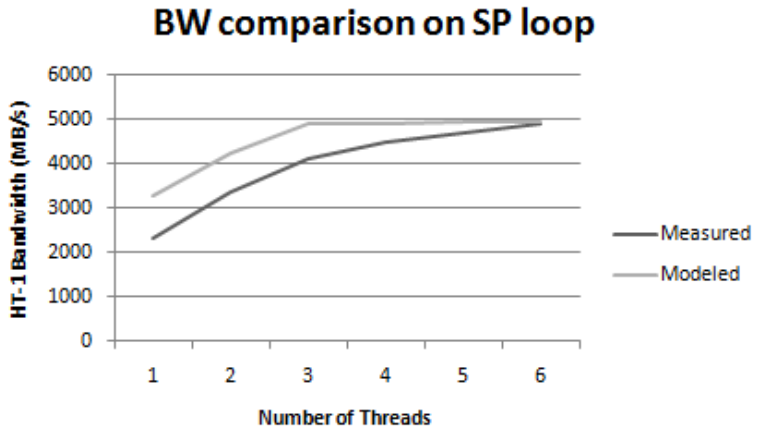


(b)

Figure 5.12: Comparison of measured and modeled bandwidths for a kernel from MG benchmark via (a) HT-2 and (b) HT-3 links.

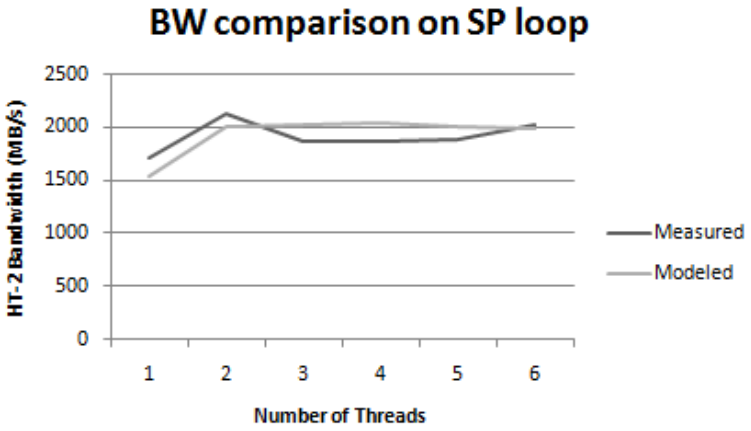


(a)

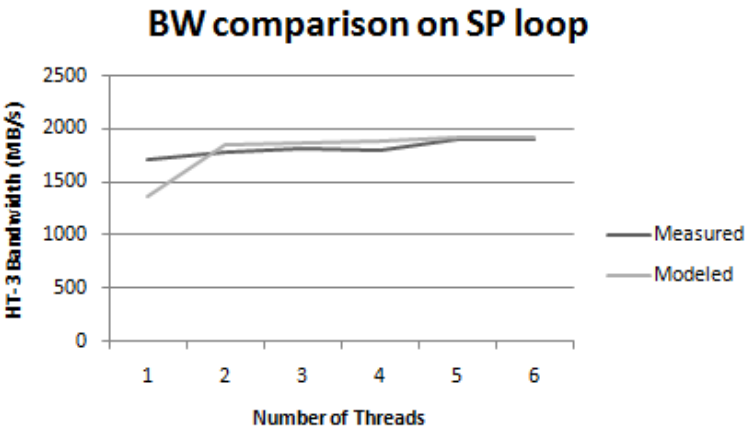


(b)

Figure 5.13: Comparison of measured and modeled bandwidths for a kernel from SP benchmark via (a) local memory and (b) HT-1 links.



(a)



(b)

Figure 5.14: Comparison of measured and modeled bandwidths for a kernel from SP benchmark via (a) HT-2 and (b) HT-3 links.

Chapter 6

Shared Cache Contention Modeling

In this chapter, we explain the importance of shared cache to the overall performance of parallel application and discuss existing research that has been done towards modeling the impact from the shared cache contention and/or sharing. We then present our approach of modeling the shared cache and discuss the results from model evaluation.

6.1 Introduction

Shared cache in multicore processors is an important hardware resource that should be utilized effectively to achieve high performance for parallel applications. It is critical to coordinate accesses by multiple threads to data that reside in shared

cache to reduce cache contentions, and to improve cache hit rates of both shared and private data of threads.

On the one hand, the shared cache is designed to be effective for data reuse by multiple cores, i.e., when a thread uses cached data previously loaded by another thread. Reusing data in shared cache decreases the number of DRAM accesses, hence improves the application performance and reduces power consumption. On the other hand, shared cache may cause severe performance degradation if large amount of cache contention and interference are incurred by uncoordinated data access of multiple cores. In fact, the effectiveness of using shared cache has been determined as the most essential factor in overall performance degradation on multicore processors [66]. Due to this reason, improving shared cache performance has become quite challenging optimization problem for parallel applications and architectures.

When running a single multithread program or multiple completely separate programs simultaneously, it is important to know how much the parallel threads or programs are exhibiting coordinated cache access behavior. In situations where their cache accesses are not properly coordinated, cache contention, trashing, and increased cache misses may occur. On the contrary, when multiple threads' cache accesses do match, e.g. accessing shared cache line with high rate, replacing cache line with low rate of conflict, they benefit each other from cache sharing, which in turn may decrease the number of cache misses and thus increase the performance of the threads. Understanding quantitatively the coordination of accesses to data residing in shared cache by multiple threads will help users and tools to optimize their code. Moreover, compilers can pass this information to the runtime to select

appropriate scheduling and data affinity policy to improve shared cache efficiency.

There has been a lot of work in the literature that focused on modeling cache performance on single processor architectures [6, 9, 20, 21, 56]. Most of these studies concentrated on determining the distribution of cache misses (compulsory, capacity, and conflict). Only recently has the modeling of cache contention and shared cache performance on multicore architectures started to receive attention [7, 10, 53, 66, 62]. The approaches to modeling cache contention have been mostly either probabilistic or simulation based.

Chandra et al. [7] propose three models to predict cache contention of co-scheduled tasks. The first two models, which are the Frequency of Access (FOA) and the Stack Distance Competition (SDC), use stack distance profile of each thread when executed sequentially. In an α - way associative LRU cache, for each cache set they have $C_{\alpha+1}$ counters such as $C_1, C_2, C_3...C_{\alpha+1}$. On each cache access, one of these counters is incremented. Thus each counter represents the access frequency to it. The $C_{\alpha+1}$ counter is incremented when the cache access results in a miss. The two models are not very accurate since they do not consider conflict misses that could happen under cache sharing when threads are co-scheduled. For this reason, the authors propose the third model which takes into account the conflict misses i.e. when a cache line that is being used by one thread is evicted by another due to the lack of cache capacity, and the former thread needs to access the same cache line again later on. The third model, *Prob Model*, is a probabilistic model that uses a circular sequence concept [7].

In [66], authors propose a simple approach of predicting cache contention using

simulation method. For that they used *setvectors* to store information about the cache set access frequency (stored in vector a) and the number of distinct cache lines accessed in the set (stored in vector d). For every memory access, authors fill out these vectors accordingly. When vectors are finished to be filled, the cache associativity level α is subtracted from each value of d, which then tells whether the number of distinct cache lines have exceeded the associativity level and would result in a cache miss or not. The results from the subtraction of α from d is stored in d' vector. Lastly, the cache set access frequency value stored in a is multiplied with values in d' for each thread and stored in vector s. The result in s tells how badly or well a thread uses that cache set when no other thread is co-scheduled. In order to predict the cache contention impact, one just needs to take a dotproduct of the two values from sx and sy for each cache set for threads x and y, and see if the result from the dotproduct is high or not. If the dotproduct result is high, it means that if threads x and y are scheduled together, it would result in a high cache contention and interference. Otherwise, if the dotproduct result is low, they would exhibit coordinated cache access behavior, and would result in less amount of cache contention. The approach in [66] is simple but effective in terms of evaluating whether the cache access behavior of two threads matches or not. However, it does not specify exactly how many cache misses would happen under co-scheduling. Moreover, their approach does not consider conflict misses; it only considers capacity misses.

The authors in [53] predict the number of compulsory and capacity misses using circular sequence profiling and stack processing techniques. Their model assumes that threads compute only homogeneous tasks such as loop iterations and share a

fully associative last level cache. Stack distance profiling is used to generate a cache trace of memory accesses by an individual thread under no co-scheduling. However with co-scheduling, the stack distance profile information becomes inadequate, thus the authors use the same concept as in [7], the circular sequence profiling, to model possible interferences from other threads. Their work models only fully associative cache and outputs only capacity misses from cache sharing by skipping the conflict misses. On the other hand, we also model set associative caches and output the total number of cache misses under co-scheduling. Fully associative caches are modeled by assuming that there is only one cache set available in the cache.

In our work, we present a new method to predict the number of cache misses that would happen when a task running on one thread is co-scheduled to run with other tasks (either homogeneous or heterogeneous), and all threads share the same last level cache. The input to our model is memory traces that can be generated by compiler analysis. The memory access trace represents a sequence of cache lines touched by a thread during some period of time. If modeling a set associative cache, the trace will represent a sequence of cache lines accessed at each cache set. When a thread is scheduled to run with another thread concurrently, the memory access traces of each thread will be combined in order to have a single complete sequence of cache lines that are being accessed by both of the threads. However, the combination of the traces is not a trivial task since at compile time we have to deal with million different combinations of sequences. By approaching to this problem from a statistical point of view, we are able to select a very small number of sequence combinations in our algorithm to calculate the number of cache misses under cache sharing. Our

evaluation demonstrates the effectiveness and accuracy of the approach for cache contention prediction. Since it is static prediction, our method is very suitable for integration into compilers' or performance estimating tools' analysis phase.

6.2 Methodology

6.2.1 Memory Access Trace Generation

Our model requires a sequence of accessed cache lines, referred to as memory access trace, for each thread and cache set during some period of time as an input. Memory access trace is generated using the same compiler analysis discussed in 4.2. We utilize the compiler analysis to collect information about the loop and its memory references. We only focus on OpenMP loops as execution block in this work. However, the compiler analysis can be extended to support parsing other types of execution blocks and generating memory access traces from them. In addition to the loop and array reference information, the compiler analysis needs to know some details about the shared last level cache of the underlying architecture such as cache size, associativity level, and cache line size. The only assumption that we make in our compiler analysis is that arrays used inside the parallel loop are declared cache aligned. The memory access trace for each thread and cache set is generated by determining the cache lines that would be accessed by each thread and placed to that specific cache set during some period of time. The compiler analysis already generates a cache line ownership list which is the list of cache lines that are accessed by a thread at specific

iteration. Then the determination of which logical cache set the cache line in cache line ownership list will go to is simply done by dividing the accessed cache line number by set associativity level α. By evaluating each thread's cache line ownership list in this way, the compiler can generate the memory access trace for each thread and the cache set.

6.2.2 Cache Miss Detection Under Cache Sharing

After we generate separate memory access traces for each thread, we would like to determine how coordinated the threads' cache access behaviors are with regards to each other, i.e. whether the cache contention and interference will be high or not, whether threads will benefit from cache sharing or not, etc. This is accomplished by estimating the number of cache misses that would happen when all memory access traces are evaluated together. In order to detect the cache misses that would happen when threads are co-scheduled, we combine separate memory access traces X_i, Y_i,Z_i ... of threads $X, Y, Z...$ into one $CombinedTrace_i$. i in X_i refers to the cache set number that the combined memory trace is generated for. For each i we have one vector S_i and a miss counter. S_i of size α, where α refers to the set associativity level of the cache, will store the different cache lines accessed during some period of time. For each cache line in S_i we also store value r which tells the logical time the cache line was accessed. The cache miss detection algorithm starts by checking whether S_i contains each cache line in $CombinedTrace$ or not.

- If S_i already contains the cache line brought from the $CombinedTrace$ list, we

update the value r of the corresponding cache line to represent the recent time it was accessed.

- If S_i does not have the cache line, we add it to S_i and update its value r. We increment the cache miss counter.

- If S_i is already full, but a new cache line from the $CombinedTrace$ needs to be added to it, we remove a cache line with the least r value from S_i and add the new cache line to S_i. We update the value r of the recently inserted cache line, and increment the miss counter.

Given X_i, Y_i,Z_i traces, one can combine these 3 traces into one combined trace in a million different ways at compile time. However, only one combination of traces will be executed at runtime. Hence, it is a real challenge to know which exact combination will be executed at compile time. Evaluating all combinations one-by-one to determine the number of cache misses would require endless days of computation. Hence, in order to quickly but accurately calculate the number of cache misses, we approach to the problem from a statistical point of view which is discussed in Simulation Case Study sub-section.

6.2.3 Simulation Case Study

In this section, we explain our approach of choosing smaller number of combinations for predicting the number of cache misses. Our case study is performed with two threads only; however the approach can be applied to a larger number of threads

without any modifications. Let $T_0 = \{ABC\}$ be the first thread's memory access sequence and $T_1 = \{XYZ\}$ be the second thread's memory access sequence. If we run two threads (T_0 and T_1) at the same time, the memory access sequence of these two co-scheduled threads will be one of the combinations of the memory accesses of T_0 and T_1. This combination can be one of the following combinations:

$$C\left(T_0, T_1\right) = \begin{cases} ABCXYZ \\ XYZABC \\ AXBYZC \\ XAYBZC \\ ... \end{cases} \tag{6.1}$$

where $C(T_0, T_1)$ $=C_{T_0,T_1}$ is the function that generates the combinations from two threads' sequences. Let $N(.)$ be a function that returns the length of the thread's memory access sequence, and $N(T_0) = m$, $N(T_1) = n$ where n and m are the length of the sequences. The total number of combinations of two threads without ordering would be $(n + m)!$. However, in separate sequences there is an order that needs to be followed such as we don't want B to appear before A in the sequence, similarly Y to appear before X. Thus, we will need to take into account the ordering of the sequences in the combinations. Then the total number of the combinations would be:

$$N(C_{T_0,T_1}) = \frac{(n+m)!}{n!m!} = M \tag{6.2}$$

Let $\mathcal{L}(.)$ be a linear function that returns the number of cache misses of a given sequence, and α_i be the number of cache misses for the i^{th} combination sequence of two threads $C_i(T_0, T_1)$.

$$\mathcal{L}(C^i_{T_0,T_1,i}) = \alpha_i \quad i = \{1 \dots M\} \tag{6.3}$$

Since, each combination is independent from the other, and $\mathcal{L}(.)$ is a linear function, then α_i $(i = 1 \dots M)$ will be i.i.d. (independent and identically distributed), which can be modeled as a Gaussian distribution [19]. This can be easily proven with the following simulation example. Let T_0 and T_1 be the sequences of random array accesses, and $n = 80$ and $m = 120$. The total number of combinations of these two threads is $N(C_{T_0,T_1}) = 1.65 \times 10^{57}$, which is very large. Generating and evaluating all of these combinations would require a lot of computation time. Instead, we randomly generated smaller number of combinations from these two thread sequences, where the total number of combinations was 2.6×10^6. We then input each combination of sequences to our prediction technique that calculated the number of cache misses that would happen for that combination. After computing the cache misses for each combination in $C(T_0, T_1)$, where we have α_i values for $i = 1 \dots 2.6 \times 10^6$, we built a histogram of the α values. Figure 6.1 depicts the simulation results. Red dots in the figure are the frequencies of the α values in $x - axis$ in the distribution. Notice that the red dots in the Figure 6.1 can be approximated with Gaussian curve (green line in Figure 6.1).

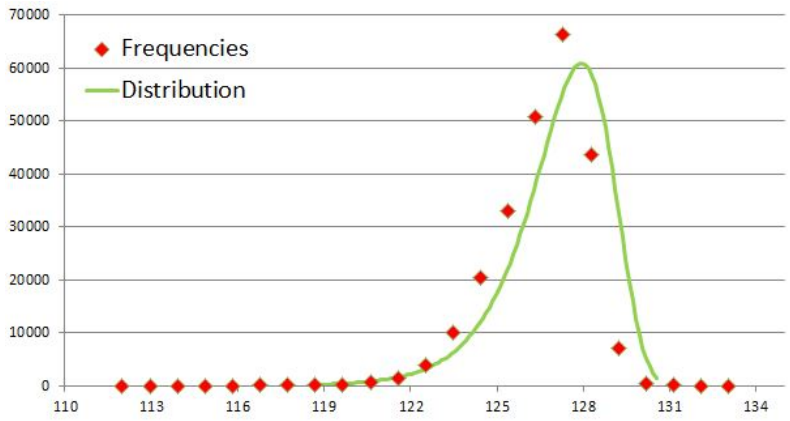


Figure 6.1: Distribution of cache miss values for $i = 1 \ldots 2.6 \times 10^6$ combinations

The distribution of cache misses α_i of the combination sequences $C_i(T_0, T_1)$ for $\forall i$ can be approximated with two parameters: mean value (μ) and standard deviation (σ) of the variables (α_i, $i = \{1 \ldots M\}$), where:

$$\mu = \frac{1}{M}\sum_{i=1}^{M}\alpha_i \quad \sigma^2 = \frac{1}{M}\sum_{i=1}^{M}(\alpha_i - \mu)^2 \tag{6.4}$$

The probability density function (pdf) [19] of the distribution of α_i for $\forall i$ is calculated as:

$$P(\alpha_i|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}e^{(\frac{-(\alpha_i-\mu)^2}{2\sigma^2})} \quad i = 1 \ldots M \tag{6.5}$$

Calculating the probability curve for $i = 1 \ldots M$ will give a curve in Figure 6.2. Figure 6.2 depicts the area percentage under the curve under different ranges. The area under the curve is computed as:

$$\mathcal{S}(\mu - \sigma \leq \alpha < \mu + \sigma) = \int_{\mu-\sigma}^{\mu+\sigma} P(\alpha|\mu, \sigma)d\alpha \quad , \tag{6.6}$$

where $\mathcal{S}(.)$ is the area.

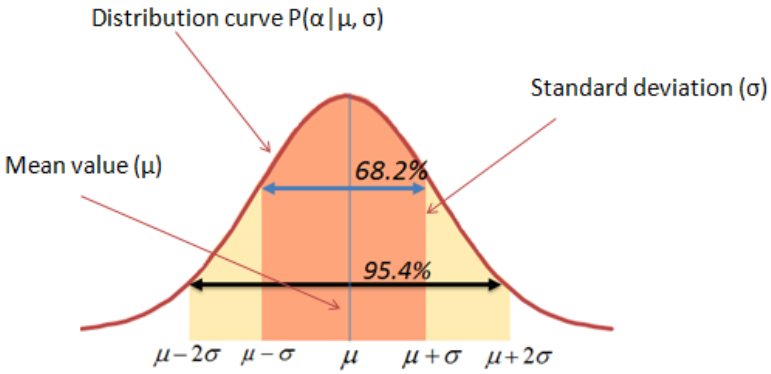


Figure 6.2: Calculated area under the distribution curve between two different intervals: ($\mu \pm \sigma$ and $\mu \pm 2\sigma$)

If $P(\alpha|\mu, \sigma)$ follows normal (Gaussian) distribution then the $\mathcal{S}(\mu-\sigma < \alpha < \mu+\sigma)$ is 68.2% of the whole area, and $\mathcal{S}(\mu - 2\sigma < \alpha < \mu + 2\sigma)$ is 95.4% of the whole area under the curve.

Using the area under the curve we can estimate the probability of predicting the number of cache misses correctly. For example, let ξ_0 be a threshold that we tolerate as an error. In other words, the value x that we predicted should be between $\mu - \xi_0 \leq x \leq \mu + \xi_0$ in order to be considered accurate. If the standard deviation σ of the distribution of the cache misses is small as in Figure 6.3a, then the probability that we estimate the number of cache misses correctly, when scheduling two threads

together, will be high, since the area ratio between the $\mathcal{S}(\mu - \xi_0 \leq x \leq \mu + \xi_0))$ and the whole area under the curve will be very high (Figure 6.3a). On the other hand, if σ of the distribution of the cache misses is high, then the probability of predicting cache misses correctly will be low, since the area ratio between $\mathcal{S}(\mu - \xi_0 \leq x \leq \mu + \xi_0))$ and the whole curve will be low (Figure 6.3b).

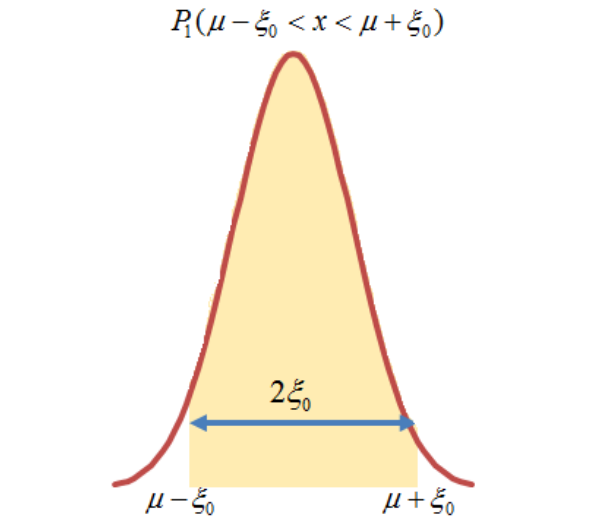
Let the parameters of normal (Gaussian) distribution function be (μ_0, σ_0) of the oversampled random variables and (μ_1, σ_1) be the normal distribution parameters of the fewer samples of random variables. The difference between mean values will be very small:

$$\|\mu_0 - \mu_1\|_2^2 \leq \varepsilon_0 \quad , \tag{6.7}$$

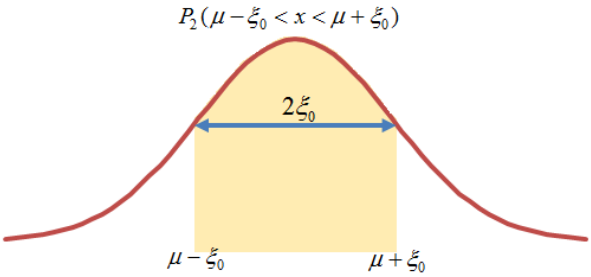
where ε_0 is a small positive value.

This can be showed with the previous example ($n = 80, m = 120$). First we computed the mean (μ) and the standard deviation (σ) values for all memory access sequence combinations where $i = 1 \ldots 2.6 \times 10^6$ (100%). Then we computed (μ,σ) for 50% of the samples, and similarly computed for 1%, 0.1% and 0.01% of the samples. Figure 6.4 depicts the results of the mean and standard deviation values for different percentages of combinations used.

Notice that in Figure 6.4 the difference between the mean values for 100% and 0.01% of the samples is very small. Figure 6.5 depicts the frequencies (red dots) of the cache misses and its estimated distribution curve (green line) for 1% of the data. It can be seen that Figure 6.5 (1% of the data) is similar to the Figure 6.1 (100% of



(a)



(b)

Figure 6.3: Probability of predicting cache miss values correctly for distributions with different standard deviation σ

Combinations used	μ	σ
100%	127.16	2.94
50%	127.16	2.91
1%	127.18	2.81
0.1%	127.35	2.75
0.01%	127.56	1.9

Figure 6.4: Comparison between mean and standard deviation values for different percentages of combinations used in the simulation

the data).

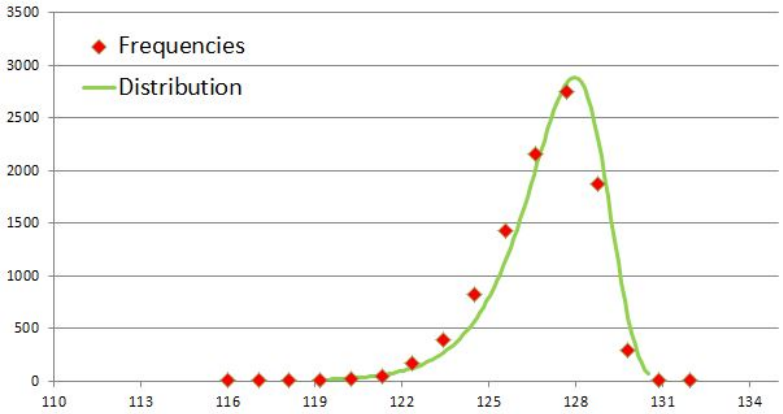


Figure 6.5: Distribution of cache miss values for 1% of $i = 1 \ldots 2.6 \times 10^6$ combinations

With the simulation case study we show that we can accurately predict the number of cache misses that would happen under cache sharing. The only challenge is a need to evaluate all memory access sequence combinations for co-scheduled threads, which will require a lot of computation time. However, we also show that if the predicted cache miss values for all combinations follow a normal distribution with a standard deviation σ and mean value μ, then with fewer samples the cache miss

values will follow normal distribution as well. Thus, instead of evaluating all combinations of the memory access sequences, we only need to evaluate few combinations and compute cache miss values for them. Then we compute the mean and standard deviation values from these cache miss values. The final value for cache misses for co-scheduled threads under cache sharing will be the mean value of the distribution. Besides, given the mean value μ, a standard deviation σ and error rate ξ_0 that we tolerate, we will also be able to tell the prediction accuracy using Equation 6.6.

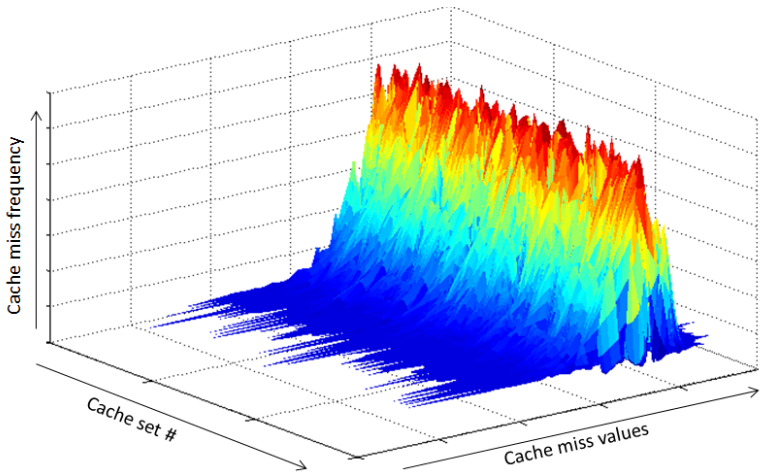


Figure 6.6: Distribution of cache misses for different cache sets

Moreover, we show that homogeneous tasks, such as loop iterations that are executed by co-scheduled threads with much larger data sets than the shared cache size, will exhibit the same cache access pattern, except that cache sets that are being accessed will vary based on the loop iteration. Thus all the cache sets will have a normal distribution of cache misses as shown in Figure 6.6. Using this information it is possible to predict the total number of cache misses for the whole cache by

evaluating few random cache sets (instead of all cache sets), by taking the average cache miss value and multiplying it by the total number of cache sets.

6.3 Results

We evaluated our cache miss prediction technique using two very widely known benchmarks, matrix multiplication and discrete fourier transform (DFT) [46]. For that, we compared the predicted number of cache misses obtained by our method against the measured cache miss values obtained using hardware performance counters.

For measuring the number of cache misses, we used OpenMP versions of the two benchmarks and measured the number of L3 (LLC) cache misses using PAPI interface [44] when both sequential and multithreaded versions of programs are executed. Experiments are performed on a system with 2.2 GHz quad core AMD Opteron processor that has 64 byte line, 32-way set associative L3 cache of 2048KB size which is shared among all four cores. The programs were compiled using GNU 4.5.0 compiler with PAPI 4.2.0 library.

Tables 6.1 and 6.2 show the comparison between measured and predicted cache miss values that we obtained for matrix multiplication and DFT kernels, respectively. The two columns (3 and 5) in both tables show the predicted number of cache misses when 250,000 and 2,000 memory trace combinations are evaluated, respectively. Since the total number of combinations for matrix multiplication and

DFT kernels were very large, we only evaluated 250,000 and 2,000 number of combinations for our experiments. Columns 4 and 6 show an error difference between the measured and the predicted cache miss values for each experimental case. Since sequential versions of programs have only one memory trace combination, only one error difference is calculated and shown in column 4. The lower the values in columns 4 and 6 are, the more accurate our prediction method is. Moreover, by comparing these two columns we evaluate the accuracy of our statistical approach to decrease the number of memory trace combinations that are being used in our prediction technique. The closer the values in both columns are, the more accurate our statistical approach is meaning that similar (or even the same) prediction accuracy still can be achieved by evaluating less number of memory trace combinations.

Both matrix multiplication and DFT kernels exhibit the same memory access pattern throughout the whole iterations of the loop. In other words, their memory access pattern does not change based on the iteration being executed. This implies that although it seems that the first several cache sets are mainly used in the first hundred (or thousands) iterations, the rest of the cache sets will be utilized in the same manner in later iterations. Due to this fact, to predict the number of cache misses for the whole cache we evaluated only few cache sets and not all of the cache sets available. The average predicted results for few cache sets were enough to estimate the total number of cache misses for the whole cache for both programs.

Results for matrix multiplication and DFT kernels given in Tables 6.1 and 6.2 show that the error difference between the predicted and the measured cache miss values varies between 12%-18% and is 16% in average, which is fairly good and

reasonable since it is very challenging to precisely estimate the number of cache misses for real systems with complex architecture. Therefore we believe that the predicted cache miss values estimated by our technique are close to the measured cache miss values, indicating that our technique can be used to predict the cache contention and/or sharing impact on multithreaded applications at compile time. Moreover, the results also show that the error differences between the measured and the predicted values stay almost the same for 250,000 and 2,000 combinations. This implies that evaluating less number of memory trace combinations does not hurt the prediction accuracy and proves that our statistical approach is technically accurate as well.

Our method of predicting cache contention impact when combined with compiler machine models, could be used by compilers to assist in optimizing code in both high-level loop transformation, and low-level instruction scheduling and code generation. For example, it would be helpful for programmers, compilers and/or performance analyzing tools to choose the optimal way of distributing iterations to threads, the number of threads to execute the loop, or to select appropriate scheduling and data affinity policy to improve shared cache efficiency. It could be used to guide a compiler when performing traditional loop transformations to decide parameters suitable for executing parallel loops on multicore architecture.

Table 6.1: Comparison of predicted vs. measured # of cache misses for matrix multiplication kernel

# of threads	Measured # of Cache Misses	Predicted # of Cache Misses (250K combinations)	Error % (250K combinations)	Predicted # of Cache Misses (2K combinations)	Error % (2K combinations)
1	113,042K	98,076K	13%	NA	NA%
2	110,045K	134,124K	18%	135,247K	19%
3	117,668K	140,317K	16%	141,432K	17%
4	110,461K	95,045K	14%	95,436K	14%

Table 6.2: Comparison of predicted vs. measured # of cache misses for dft kernel

# of threads	Measured # of Cache Misses	Predicted # of Cache Misses (250K combinations)	Error % (250K combinations)	Predicted # of Cache Misses (2K combinations)	Error % (2K combinations)
1	424K	510K	17%	NA	NA%
2	1,111,029K	950,501K	14%	929,361K	15%
3	305,406K	370,304K	18%	371,577K	18%
4	7,550K	8,574K	12%	8,658K	13%

Chapter 7

Conclusion

In this dissertation, we described how existing compilers' cost models can be extended in order to consider performance critical features that come with multicore and many-core architectures such as false sharing, contention for shared memory bandwidth, and shared last level cache. We proposed compile time models for each of these features that estimate the performance impact range it would have on application performance. Firstly, we described our compile-time cost model for false sharing and discussed how to use the defined model to detect and estimate the performance impact of false sharing on parallel loops. The model estimates the total number of false sharing cases that occur throughout the loop, and computes the overhead cost incurred by false sharing to the whole execution of the loop. Moreover, we describe our false sharing prediction model that predicts the total false sharing cases by evaluating much fewer number of iterations, for the purpose of reducing the modeling time. Secondly, we presented our compile time off-chip memory bandwidth model

to estimate the bandwidth performance of OpenMP parallel loops. We used the statistical polynomial curve fitting technique on a set of bandwidth measuring data obtained through experiments by the modified STREAM benchmark. This model could be used by the compiler and performance tools to predict when the sustainable memory bandwidth of the system will be reached by the application during execution, and to determine an optimal number of threads that should be configured to execute a specific parallel loop according to its memory reference pattern. Lastly, we presented a new method to predict number of cache misses that would happen due to cache sharing and/or contention at compile time. The method utilizes compiler analysis techniques to generate memory access traces for each thread under no cache sharing. Then a small number of combinations of threads' memory access traces are randomly selected among all combinations as an input to predict cache misses of the threads using the proposed method.

All of our models have been evaluated by a set of widely known OpenMP kernels. The results obtained are very promising, and we believe that these three models can be used to accurately estimate the performance impact (either bad or good) and thus guide compilers', performance analyzing or tuning tools' optimizations.

Bibliography

[1] D. Agarwal, W. Liu, and D. Yeung. Exploiting application-level information to reduce memory bandwidth consumption. In *Proceedings of the 4th Workshop on Complexity-Effective Design*, 2003.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[3] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems (TOCS)*, 28(4):8:1–8:45, 2010.

[4] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 3–18, 1993.

[5] J. R. Cannon. The one-dimensional heat equation. *Encyclopedia of Mathematics and Its Applications*, 1984.

[6] C. Cascaval, L. D. Rose, D. A. Padua, and D. A. Reed. Compile-time based performance prediction. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 365–379, 2000.

[7] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340 – 351, 2005.

[8] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, Cambridge, Massachusetts and London, England, 2008.

[9] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 286–297, 2001.

[10] X. E. Chen and T. Aamodt. Modeling cache contention and throughput of multiprogrammed manycore processors. *IEEE Transactions on Computers*, 61(7):913–927, 2012.

[11] J.-H. Chow and V. Sarkar. False sharing elimination by selection of runtime scheduling parameters. In *Proceedings of the 26th International Conference on Parallel Processing*, pages 396–403, 1997.

[12] I. D. Coope. Circle fitting by linear and nonlinear least squares. *Journal of Optimization Theory and Applications*, 76(2):381–388, 1993.

[13] G. Daci and M. Tartari. A comparative review of contention-aware scheduling algorithms to avoid contention in multicore systems. In *Proceedings of the Third International Conference on Trends in Information, Telecommunication and Computing*, volume 150 of *Lecture Notes in Electrical Engineering*, pages 99–106. Springer New York, 2013.

[14] J. Diamond, M. Burtscher, J. McCalpin, B.-D. Kim, S. Keckler, and J. Browne. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 32–43, 2011.

[15] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–241, 1999.

[16] C. Ding and K. Kennedy. The memory bandwidth bottleneck and its amelioration by a compiler. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, pages 181–189, 2000.

[17] J. Dongarra, D. Gannon, G. Fox, and K. Kennedy. The impact of multicore on computational science software. *CTWatch Quarterly*, 3(1):3–10, 2007.

[18] N. Draper and H. Smith. *Applied Regression Analysis*. Wiley, 1998.

[19] M. Evans, N. Hastings, and B. Peacock. *Statistical Distributions*. Wiley, 2000.

[20] B. B. Fraguela, R. Doallo, and E. L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 221–231, 1999.

[21] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):703–746, 1999.

[22] S. M. Günther and J. Weidendorfer. Assessing cache false sharing effects by dynamic binary instrumentation. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*, pages 26–33, 2009.

[23] T. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, 1994.

[24] A. Kayi, T. El-Ghazawi, and G. B. Newby. Performance issues in emerging homogeneous multi-core architectures. *Simulation Modelling Practice and Theory*, 17(9):1485 – 1499, 2009.

[25] A. Kayi, Y. Yao, T. El-Ghazawi, and G. Newby. Experimental evaluation of emerging multi-core architectures. In *Proceedings of the Parallel and Distributed Processing Symposium*, pages 1–6, 2007.

[26] R. P. LaRowe, C. S. Ellis, and V. Khera. An architecture-independent analysis of false sharing. Technical report, Duke University, 1993.

[27] H.-J. Lee, W.-C. Cho, and E.-Y. Chung. Analytical memory bandwidth model for many-core processor based systems. *IEICE Electronics Express*, 9(18):1461–1466, 2012.

[28] J. Levesque, J. Larkin, M. Foster, J. Glenski, G. Geissler, S. Whalen, B. Waldecker, J. Carter, D. Skinner, H. He, H. Wasserman, and J. Shalf. Understanding and mitigating multicore performance issues on the amd opteron architecture. Technical report, Lawrence Berkeley National Laboratory, 2007.

[29] C. Liao and B. M. Chapman. Invited paper: A compile-time cost model for OpenMP. In *Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2007.

[30] C. Liu. False sharing analysis for multithreaded programs. Master's thesis, National Chung Cheng University, 2009.

[31] F. Liu, X. Jiang, and Y. Solihin. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[32] L. Liu, Z. Li, and A. H. Sameh. Analyzing memory access intensity in parallel programs on multicore. In *ICS*, pages 359–367, 2008.

[33] Z. Majo and T. R. Gross. Memory system performance in a numa multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, pages 12:1–12:10, 2011.

[34] A. Mandal, R. Fowler, and A. Porterfield. Modeling memory concurrency for multi-socket multi-core systems. In *ISPASS*, pages 66–75, 2010.

[35] J. Marathe and F. Mueller. Source code correlated cache coherence characterization of OpenMP benchmarks. *IEEE Transactions on Parallel and Distributed Systems*, 18:818–834, June 2007.

[36] J. Marathe, F. Mueller, and B. R. de Supinski. Analysis of cache coherence bottlenecks with hybrid hardware/software techniques. *ACM Transactions on Architecture and Code Optimization (TACO)*, 3(4):390–423, 2006.

[37] P. Marchal, J. I. Gómez, and F. Catthoor. Optimizing the memory bandwidth with loop fusion. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 188–193, 2004.

[38] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: Analyzing memory system bottlenecks in programs. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, 1992.

[39] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, 1995.

[40] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, 1996.

[41] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems*, pages 153–166, 2010.

[42] R. M. Mohideen and V. Sankaranarayanan. An analytical model for optimum off-chip memory bandwidth partitioning in multicore architectures. In *Proceedings of the Second International Conference on Computer Science and Information Technology (ICCSIT)*, 2012.

[43] S. K. Moore. Multicore is bad news for supercomputers. *IEEE Spectrum*, 45(11):15–15, 2008.

[44] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[45] J. Parkhurst, J. Darringer, and B. Grundmann. From single core to multi-core: Preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*, pages 67–72, 2006.

[46] W. H. Press, W. T. Vettering, S. A. Teukolsky, and B. P. Flannery. Fourier transform of discretely sampled data. *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, pages 494–498, 1989.

[47] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, 2006.

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

[49] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[50] A. Sandberg, D. Eklov, and E. Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[51] D. Schuff, B. Parsons, and V. Pai. Multicore-aware reuse distance analysis. In *IPDPS Workshop on Performance Modeling, Evaluation, and Optimization of Ubiquitous Computing and Networked Systems*, pages 1–8, 2010.

[52] A. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh. Parallelism via multithreaded and multicore cpus. *Computer*, 43(3):24–32, 2010.

[53] F. Song, S. Moore, and J. Dongarra. L2 cache modeling for scientific applications on chip multi-processors. In *Proceedings of the International Conference on Parallel Processing*, pages 51–58, 2007.

[54] L. Tongping and E. D. Berger. Precise detection and automatic mitigation of false sharing. In *Proceedings of ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications (OOPSLA/SPLASH)*, pages 3–18, 2011.

[55] B. Venu. Multi-core processors - an overview. *CoRR*, abs/1110.3535, 2011.

[56] X. Vera and J. Xue. Let's study whole-program cache behaviour analytically. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 175–186, 2002.

[57] R. Wang, L. Chen, and T. M. Pinkston. An analytical performance model for partitioning off-chip memory bandwidth. In *Proceedings of the IPDPS*, pages 165–176, 2013.

[58] Y. Wang, Y. Cui, P. Tao, H. Fan, Y. Chen, and Y. Shi. Reducing shared cache contention by scheduling order adjustment on commodity multi-cores. In *IPDPS Workshops*, pages 984–992, 2011.

[59] B. Wicaksono, M. Tolubaeva, and B. Chapman. Detecting false sharing in OpenMP applications using the DARWIN framework. In *Proceedings of International Workshop on Languages and Compilers for Parallel Computing*, pages 283–297, 2011.

[60] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture*, pages 274–286, 1996.

[61] X. Wu and V. E. Taylor. Performance modeling of hybrid mpi/openmp scientific applications on large-scale multicore cluster systems. In *CSE*, pages 181–190, 2011.

[62] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache contention and application performance prediction for multi-core systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 76–86, 2010.

[63] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 89–102, 2009.

[64] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 27–38, 2011.

[65] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. *SIGPLAN Not.*, 45(3):129–142, Mar. 2010.

[66] M. Zwick, F. Obermeier, and K. Diepold. Predicting cache contention with setvectors. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, pages 244–251, 2010.