# SHARING AWARE SCHEDULING ON MULTICORE SYSTEMS

by

Ali Kamali
B.Sc., Sharif University of Technology, 2008

A Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the School
of
Computing Science

## APPROVAL

| | |
|---|---|
| **Name:** | Ali Kamali |
| **Degree:** | Master of Science |
| **Title of Thesis:** | Sharing Aware Scheduling on Multicore Systems |

**Examining Committee:** Dr. Ke Wang
Chair

Dr. Alexandra Fedorova, Professor, Computing Science
Simon Fraser University
Senior Supervisor

Dr. Greg Mori, Professor, Computing Science
Simon Fraser University
Supervisor

Dr. Hao Zhang,
Professor, Computing Science
Simon Fraser University
Examiner

**Date Approved:** August 6, 2010

# Abstract

Multicore processors are becoming more and more widespread, specially in the server market. Data centers can harvest the power of multicore systems once proper scheduling methods are comprehended. Current operating systems do not consider all characteristics of the applications that are being scheduled and therefore cannot make optimal scheduling decisions. This will waste the power of multicore systems and increase the costs of a data center. The idea explained in this thesis is to solve one of the problems of scheduling on multicore systems. Using the methods introduced in this thesis, operating systems can detect data sharing between different threads of a multithreaded application and make better scheduling decisions. Sharing aware scheduling can improve the performance of applications by up to 42%. The scheduler can detect data sharing dynamically just by monitoring hardware performance counters.
**Keywords:** Multicore systems, sharing aware, scheduler, CMP, operating system, MESI, data sharing, snoop, hardware performance counters

*To my family, Leila, and all my friends in Vancouver*

*"I can no other answer make, but, thanks, and thanks."*

*— William Shakespeare*

# Acknowledgments

I would like to thank my senior supervisor, Sasha Fedorova, who helped me through my studies and guided me in the realms of multicore systems! I would also like to thank my colleagues at SFU.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Increasing clock speed of microprocessors is reaching diminishing returns. Newer processors are increasing the number of cores per chip and multicore systems are becoming more and more widespread. Operating systems however, are not advancing with the same pace. Optimal scheduling on multicore systems is still an open problem.

In order to understand why scheduling on multicore systems is a challenging task first we need to understand how multicore systems work. Multicore systems have more than one core on each chip and each system can contain one or many chips. These cores need to access the main memory which is quite slow compared to the processing power of each chip. Cores usually waste a lot of cycles waiting for the data to be fetched from the memory. To address this problem multiple level of data and instruction caches were introduced. Each core has a private L1 data and instruction cache. Depending on the architecture each core can either have a private L2 cache or share it with other cores. Some newer architectures have introduced a L3 cache which is always shared among all the cores on each chip. On each access to the memory, local caches are searched first and if the data is not present in those caches a request will be sent to the memory to fetch the data. If the data is actually present in any of the caches then there will be no need to send a request to the memory, and since delay of accessing caches is much less than the delay of accessing the main memory, cores will save some cycles.

In order to maintain a coherent system cores need to talk to other cores on the same chip and the cores on other chips so that all of them access up-to- date data. Each time a core modifies some data it needs to notify all other cores that the data was changed. This is done via cache coherency protocols such as the MESI protocol. Sending coherency messages cross-chips is very expensive and affects the overall performance of the processors[12, 11, 7, 4]. Cross-chip coherency messages can be avoided by placing the threads on the same chip so that the threads share the last level cache.

The size of caches on each chip is limited. Cores compete with each other for cache resources. There is also only one memory controller per chip and cores need to use the memory controller each time they access data in the memory. These are the common problems that applications that run on multicore systems face. The coherency messages that the cores send to each other will also affect the performance of applications [5].

An operating system scheduler needs to consider these bottlenecks and make intelligent decisions on how to place the threads on different cores. The way the scheduler decides to place the threads on the cores will impact the overall performance of the system. Different applications with different characteristics will run on the system and the scheduler needs to monitor the behavior of these applications and make its decisions based on that. To facilitate this process for schedulers, modern processors provide invaluable information about the performance of each core via hardware performance counters. Hardware performance counters are special purpose registers put into the processors to reflect the activities of each core.

In this thesis we focus on reducing the number of messages due to the coherency protocol. We exploit some of the performance counters to gain knowledge about the amount of data sharing among threads. Using this knowledge we design a scheduler that intelligently schedules the threads that are sharing data onto a single chip so that the threads share their data on the shared last level cache. Hence, they no longer need to send messages across chips to maintain their coherency. We call this behavior being sharing-aware, and the placement that the scheduler chooses is a sharing-aware placement. Just using this technique we show that the runtime of multithreaded applications that share data can be improved by up to 42%.

# Chapter 2

# Related Work

The work by Tam et al. [8] is the closest to the work that has been done in this thesis. In their work, the authors use hardware performance counters to analyze the stall cycles of each core. If cross-core communication is among the sources of stalls they monitor all threads in the system to gather information about the data regions accessed by the threads that caused cross-chip communication. That is done with data sampling capabilities of hardware performance counters. Then they cluster the threads that access the same data. Their scheduler tries to schedule the threads close to each other based on the cluster groups. Although this work showed improvements of up to 7% in some workloads, their method cannot be used on Intel processors and does not allow the schedulers to predict the benefits from sharing-aware placement.

Bellosa and Steckermeier [1] introduce a new scheduling architecture with the goal of reducing the number of cache misses. They started by using hardware performance counters to identify data sharing between the threads. However due to the limitation of hardware performance counters and the delay of acessing them at the time of their work they could not obtain good results. Weissman [11] requires users to identify the shared regions. Based on the information that the users provide and the cache miss information data gathered by hardware performance counters, he designs a portable parallel system that eliminates a lot of cache misses.

Thekkath and Eggers [9] hypothesized that they can reduce cache interference and improve the overall performance by co-locating the threads that share data on the same processor. They used a parallel tracing tool called MPtrace to gather information about sharing characteristics of their workloads. To test their hypothesis the authors compared a variety of thread placement algorithms via simulation. Simulation results showed that compulsory and invalidation misses did not decrease with any of the thread placement algorithms. They could not show any benefit from co-locating the threads on the same processor. They concluded that they could not show any benefit because of the way their benchmarks were accessing the memory and the fact that the data sharing between the threads was insignificant. In this thesis we do not use offline analysis, we rely on hardware performance counters to obtain online data and we dynamically make our decisions.

Zhuravlev et al. [13] tried to minimize the number of cache misses by reducing the contention for accessing the shared resources, such as memory controller and shared caches. They designed a new scheduler that uses a heuristic algorithm based on the cache miss rate of applications and they showed that the scheduler performs within 2% of the optimal scheduler. In their work, however, the authors only studied the applications that suffer from running on a shared cache. They did not consider the fact that some applications benefit from running on shared caches. Similarly Chandra et al. [2] used stack distance profile and stack distance competition to predict the effects of sharing a cache between cache intensive applications. In this thesis we do not study the effects of contention separately. We introduce a model that accounts for the effects of cache contention while providing the scheduler with the information about data sharing.

Sridharan et al. [7] proposed that threads must be migrated to ”lock hot” processors. They argued that by migrating threads to such processors threads can reuse the critical section data that is already in the processor’s cache, and therefore synchronization can be done faster. To achieve their goal they modified the user level libraries to help them identify user level locks. After finding out which locks are being accessed the most the threads that are accessing them will be migrated to the same processor to exploit the shared last level cache. That approach requires modifications to the application libraries. Our approach required no modification to the applications.

Zhang et al. [12] studied the effects of cache sharing on multicore systems. The authors claim that the placement of threads does not have an impact on the overall performance of PARSEC, a recently released benchmark for multicore systems. After analyzing the access patterns of the benchmarks, the authors concluded that the benchmarks would not benefit from intelligent placement because of three reasons. First, the data sharing is uniform among threads, meaning that all threads share the same amount of data with each other, and it is difficult (or impossible) to place all threads on the same chip. Second, the amount of shared data is insignificant since data is partitioned. And third, the working set of the benchmarks is usually larger than the cache size. The authors modify some benchmarks from PARSEC so that the data sharing is no longer uniform. With the new set of benchmarks they showed that a sharing aware scheduler can improve the performance by up to 36%.

Torellas el al. [10] analyzed six applications and studied the effects of false sharing on the miss rate. The authors proposed changes to the compilers so that they would place the data structures with a different layout in the caches to prevent false sharing. Eggers and Katz [4] studied the effects of sharing on the performance of the bus and on the performance of parallel applications. Their study showed that parallel programs produce more cache misses, and better compiler technology and better software development techniques can improve the performance of parallel applications by organizing the data differently in the memory.

Chen et al. [3] evaluated the performance of two greedy schedulers. Parallel Depth First (PDF), a scheduler designed for constructive cache sharing, and a traditional scheduler. Their study showed that using PDF off-chip traffic is reduced and runtime improvements of up to 60% can be gained. Jeleel et al. [6] proposed the use of binary instrumentation to characterize cache performance. Instead of using trace-driven simulators, the authors used Pin (a binary instrumentation tool by Intel) to study the performance of the memory. Their simulator, CMP$im, was a parallel memory system simulator that could model any kind of cache hierarchy. Using CMP$im the authors could measure the amount of data sharing in detail. Their analysis however had to be done offline and therefore their method cannot be used in an online scheduler.

# Chapter 3

# Basic concepts

This chapter describes different types of systems, and some of the reasons behind choosing the correct hardware performance counters. It begins with describing how cores and caches are connected, what is a local hit, a local miss, a remote hit, and a remote miss. Then it will explain what types of misses there are, and why we are focusing on one particular type of miss in this thesis.

## 3.1 Multicore Systems

The architecture of multicore systems has been improved a lot since the introduction of the first system in 2001. Different vendors follow different methodologies. These methodologies had a great impact on the performance of the CPU. In this thesis we studies two different common architectures.

### 3.1.1 AMD Cache Architecture

AMD introduced on chip L3 caches and a private L2 cache for each chip (as shown in figure 3.1).

L2 and L3 caches on AMD systems are victim caches (they are not inclusive). Whenever there is a cache miss[1], the CPU brings the data directly to the L1 cache, skipping the others.

[1]A cache miss is when a core tries to access a data, and it cannot find the data in its cache. The core misses its cache and needs to fetch the data from the main memory.

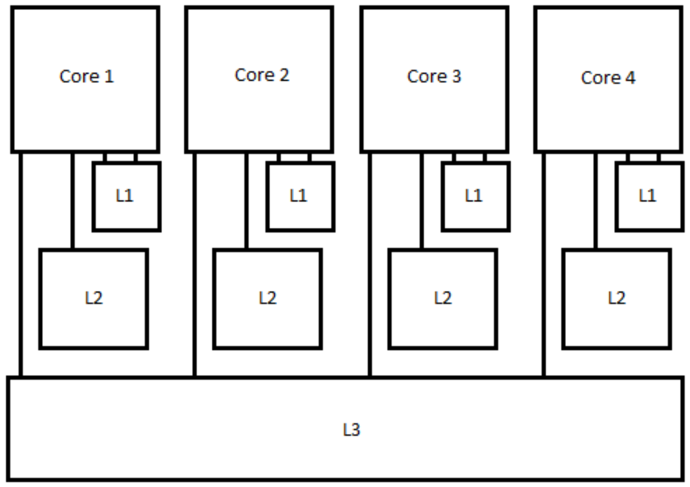

Figure 3.1: AMD Cache Architecture

The data will only be loaded to the L2 cache when it is evicted from the L1 cache, and it is loaded into the L3 cache only when it is evicted from the L2 cache.

### 3.1.2 Intel Xeon and Intel Nehalem Cache Architecture

First dual core CPUs only had L1 and L2 caches. There was a L1 cache for each core and a L2 cache was shared among the two cores on the chip (figure 3.2).

In the later designs a L3 cache was introduced and added to the same bus. In some quad core designs, two cores would share the same L2 cache, and all the cores would share the L3 cache. The limitation of this design was the memory bus for accessing the L2 cache and the memory controller. When the traffic goes up, the performance of the bus limits the performance of the CPUs.

By introducing the Nehalem architecture, Intel redesigned its CPUs and introduced private L2 caches for cores and a unified L3 cache shared for all of the cores (figure 3.3).

This design reduces the latency for accessing the caches. L3 caches are inclusive in the Intel design. It means that if data is not present in the L3 cache, it is assured that the data cannot be present in L2 and L1 caches above, and if data is present in a L2 or a L1 cache, it can be found in the L3 cache as well.

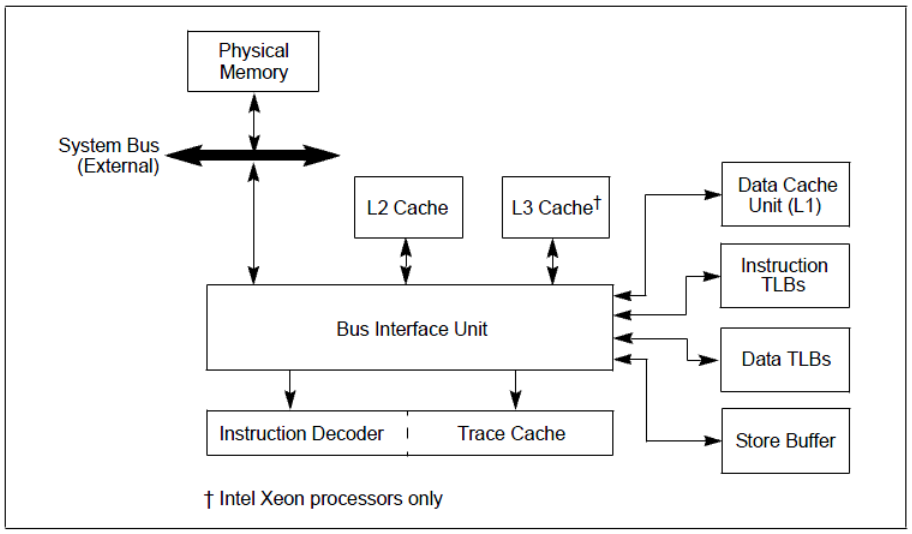

Figure 3.2: Shared L2 Cache

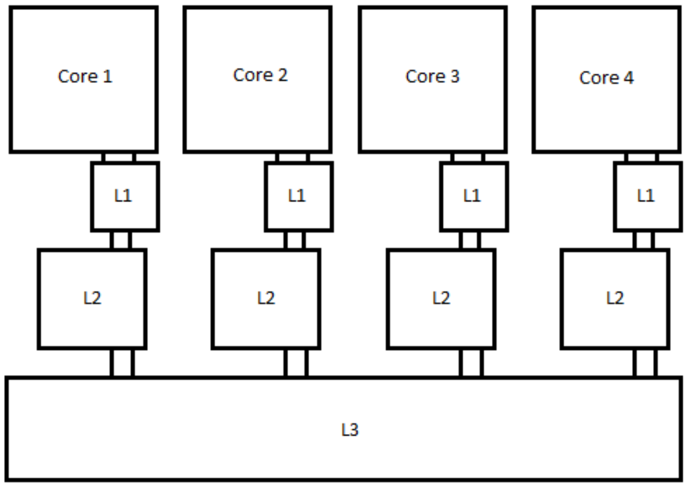

Figure 3.3: Intel Nehalem Architecture

## 3.2 Types of Cache Misses

For reducing the number of cache misses we need to find out the source of a cache miss:

### 3.2.1 Conflict Misses

Conflict misses happen when multiple memory locations get mapped to the same line in the cache. This type of cache miss can be resolved by increasing the cache size, and by increasing the associativity.

### 3.2.2 Capacity Misses

Capacity misses are due to the size of the cache. The cache is not big enough to hold the data while it is being referenced again. This type of cache miss can be resolved by increasing the cache size.

### 3.2.3 Compulsory Misses

Compulsory misses happen with the first reference to the requested data. The data is simply not in the cache and the CPU needs to load it for the first time. Prefetching helps to some degree. This type of cache miss is not a major problem.

### 3.2.4 Coherency Misses

Coherency misses only happen in multiprocessors and multicore systems. In single core processors when data is loaded into the cache, it will remain the same until that core changes it. In multicore processors however, the data can be modified by other cores while it is in the cache of a specific core. Since the data is modified by other cores, it is no longer valid and should not be used by the core that does not have a recent copy of the modified data. The data in the old cache is invalidated, therefore the next reference to that data will miss the cache. This type of misses are because of coherency messages and are called coherency misses. To resolve this problem, whenever a core modifies a data in its private cache, it invalidates the data in all the other caches. On the next reference to that data, the other cores will miss in their cache and have to reload it from either the main memory or from other caches.

The main focus of this thesis is on reducing the number of coherency misses. To understand coherency misses better, we need to understand how the coherency protocol works. The most common protocol used in the recent architectures is the MESI protocol. Each vendor has made a slight modification to the MESI protocol so that it suits their architecture.

## 3.3 The MESI Protocol

The MESI (modified, exclusive, shared, invalid) cache protocol maintains consistency between cache lines of different processors. Each cache line is in either modified, exclusive, shared, or invalid state. The state of a cache line is updated whenever one of the cores try to access that cache line:

- **Invalid:** A cache line is in invalid state when the data in that cache line does not represent any data in the memory.
- **Exclusive:** A cache line will be tagged as exclusive when a core loads the data into its cache for the first time and the data is not present in the private cache of any other core, and is not present in the shared cache of other chips.
- **Shared:** When a cache line is in shared state, it means that the cache line represents the latest value for that memory location, and the data might be found in the private cache of other cores, or might be present in the shared cache of other chips.
- **Modified:** A modified cache line is a cache line that was in exclusive mode, but the data has been modified by the core that owned the cache line and it no longer represents the data that is in the memory. The value of a cache line that is in modified mode must be written back to memory whenever the cache line is evicted or its state is changed.

Table 3.1 presents a summary of the MESI protocol and explains how the MESI protocol decides about the state of each cache line.

The important thing to notice in this protocol is that whenever a core that does not own a cache line (meaning that the cache line is not in exclusive or shared state) tries to modify it, it will result in sending an invalidation message to the other caches. This will result in all the other cores missing the data in their caches on their next reference.

Table 3.1: The MESI protocol state machine

| Old MESI state | Event | What to do | New state |
|---|---|---|---|
| Invalid | Read miss, data found in other caches | Load cache line | Shared |
| | Read miss, data not found in other caches | Load cache line | Exclusive |
| | Write miss | Invalidate data in other caches, load cache line, modify cache line | Modified |
| Shared | Read hit | | Shared |
| | Write hit | Invalidate data in other caches | Modified |
| | Read snoop hit | | Shared |
| | Invalidate snoop hit | Invalidate the cache line | Invalid |
| Exclusive | Read hit | | Exclusive |
| | Write hit | | Modified |
| | Read snoop hit | | Shared |
| | Invalidate snoop hit | Invalidate the cache line | Invalid |
| Modified | Read hit | | Modified |
| | Write hit | | Modified |
| | Read snoop hit | Write the cache line back to memory | Shared |
| | Invalidate snoop hit | Write the cache line back to memory | Invalid |
| | Eviction | Write the cache line back to memory | Invalid |

Because of the way that the MESI protocol works, cores will send snoop messages to each other to find the status of data in their caches. For example whenever there is a cache miss, there will be an snoop message (read snoop) sent to all the other caches. All caches will listen to the snoop messages. They will decide what to do based on their current state and the snoop message. If other cores have the data, then the cache line will be marked as shared.

The snoop messages cause a lot of traffic on the bus. One of the reasons that Intel and AMD decided to disconnect the L3 cache from the shared bus for the L2 cache, was to reduce the contention for the bus. The caches do not compete with each other to access the cache anymore. Different studies have shown the effect of bus contention on the performance of the CPU.

## 3.4 Read and Write Sharing

Let's see how an application's performance can change depending on how it is being scheduled on the cores. An application might either benefit from read or write sharing.

**Read sharing** Read sharing happens when one core prefetches some data for the other cores. For example assume that two threads are going to read an array and do some arithmetic operations on it.

If the cores do not share a cache, then the first reference to a data will miss the cache of the core that is trying to access the data. When the other core tries to access the data, it will miss its cache as well. In this case the missing data can either be loaded from the other cache via interconnects (cache to cache transfer), or it can be loaded from the memory.

If the cores share a cache the first reference to the data will miss the cache and will cause the data to be loaded into the cache. When the other core tries to access the same data, it will no longer miss the cache since the data has already been loaded to the cache.

In practice however, read sharing is not something that applications can benefit from. We believe so because the working set of applications either fits into the shared cache or it does not. If it fits into the shared cache, then co-locating the threads onto a shared cache might actually hurt their performance since they have to split the cache for the part of the data that is not shared. It is better to schedule them on separate chips, each thread will load all its data into its own last level cache and will not miss the data anymore. If the working set does not fit into the shared cache (which is the case most of the times), the threads need to be synchronized to be able to benefit from read sharing. Because once one thread loads some data into the cache, the other thread might access it so late that the data is evicted from the cache. In this case the threads will only benefit from sharing if they agree on which parts of the data they are going to read. This agreement requires synchronization which will hurt the performance of the threads.

**Write sharing** This type of sharing happens when two cores modify the same line in the cache. As an example assume that two threads are going to increase a shared counter.

If the cores do not share a cache, the first core that tries to write to the counter will probably miss the cache, because it needs to read the value of the counter first in order to increase it. When the other core tries to increase the value, it will miss the cache as well. The data will be loaded into the second cache. When the second core increases the value of the counter it will invalidate the data in the first cache. Consequently when the first core tries to increase the counter, it will again miss the cache since its data has been invalidated.

If the cores share a cache there will be a miss only for the first reference to the data. The first miss is a compulsory miss. The first core will load the data into the shared cache and it will modify it. When the second core tries to increase the value of the counter it will no longer miss the cache since the data is already there. It will just increase the value of the counter.

### 3.4.1 How does MESI protocol fit here?

Let's examine each case in detail and see how the MESI protocol works.

- **Read sharing on separate caches** The first core will try to read the data. The data is not present in its cache. This core sends a snoop message to the other caches to see if there is a copy anywhere. If there is a copy in any of the other caches, those copies will be marked as *Shared* (remember the MESI protocol line states). If there is a copy in another cache and it is in *Modified* state, that copy will be written back to the memory first, and then it will be marked as *Shared*.

  When the second core tries to access the same data, the exact same process will happen. A snoop message will be sent on the bus and the copy of the data in the other cache will be marked as *Shared* if it is not in that state. Depending on the cache-to-cache transfer latency the missing cache might either load the data from the other caches, or load the data directly from the memory.

- **Read sharing on a shared cache** Similar to the case of separate caches, the first core will try to read the data and will miss the cache. A snoop message will be sent to all the other caches to see if there is a copy.

  The second core however, will not miss the cache when it tries to access the data. Since the data is already in the shared cache and is in either *Shared* or *Exclusive* state, no snoop messages will be sent.

- **Write sharing on separate caches** Write sharing in this case is really expensive. Let's assume that the cores will just write to the shared value, meaning that they will not read the value first in order to change it. On the first write access the first core sends out a RFO[2] message. This message will invalidate the data in all other caches and will bring the data to the cache of the first core in an *Exclusive* state. If there is a *Modified* copy of the data in other caches, it will be written back to the memory. After the data is loaded to the cache, the core can modify the value of the data in its cache. This will cause the cache line to go to the *Modified* state. Consecutive writes to that location by the first cache will not produce any miss or snoops.

[2]Read For Ownership

When the second core tries to modify the data, it will send a RFO message, causing the first cache to write the data back to the memory and invalidating the cache line. Then the data will be loaded into the second cache and will be modified by the second core. Since the data is invalidated in the first cache, the next write to the data will result in the first core sending out another RFO message invalidating the data in the second cache.

- **Write sharing on a shared cache** When the cores are sharing a cache, everything is much simpler. The first access will send a RFO message to all the caches. Consecutive modifications by either of the cores will not result in a miss or a RFO message. The data will just be modified in the shared cache. The data will be written back to the memory when it is being evicted due to a conflict, or due to a RFO message from another cache.

## 3.5 Sources of Coherency Misses

All operations that end in a coherency miss can be classified as following.

- **False Sharing** In most modern architectures cache lines are 64 bytes[3]. It means that regardless of the type of the data that a core is trying to access (4 byte integer, or an 8 byte double) 64 bytes of data will be loaded into the cache. False sharing happens when two cores try to access different parts of a line in the cache. Consider the following example:

```
int first_variable;
int second_variable;

void first_core_worker(){ ... } // modifies first_variable

void second_core_worker(){ ... } // modifies second_variable
```

[3]On some architectures it is 128 bytes

Logically each core is modifying its own data and there is no sharing between the threads. However when this application is compiled and running, those two variables may be mapped to a single cache line. When each core writes to its variable it will cause the value (and hence the cache line) to be invalidated in all the other caches. Since the other variable is on the same cache line, it will be invalidated in all the caches. If the cores are not sharing a cache, the other core will miss the cache when it tries to access its data.

In this case the programmer unintentionally created a sharing. This type of sharing is called false sharing. This is a very common mistake among programmers (or compilers). The cache misses due to false sharing can be avoided by profiling the application and avoiding the false sharing by aligning each variable to a different cache line. Of course this solution is not practical when the source code is not available.

- **True Sharing** True sharing happens when the value that is being invalidated in the other caches is actually the value that is being used by all the cores. The following example is a case of true sharing:

  ```
  int shared_variable;

  void first_core_worker(){ ... } // modifies shared_variable

  void second_core_worker(){ ... } // modifies shared_variable
  ```

  The only way to avoid true sharing is to change the algorithm of the application, although changing the algorithm is not always feasible.

Note that in both cases we have write sharing. As we will see later, cache misses due to both true and false sharing can be avoid using a shared cache.

## 3.6 Why AMD systems do not benefit from sharing

While we were studying the cache architecture of AMD systems we noticed that they might not be able to benefit from sharing at all. On AMD architecture, after a cache miss the data will be directly loaded into the L1 cache, bypassing L2 and L3 caches. When two threads running on a shared cache on a AMD system, the first reference to the data by the first core will result in the data being loaded into the private L1 cache of the first core. When the second core accesses the same data it will miss the cache as well, because the data is not present in the shared cache. It will be present in the shared cache whenever it is evicted from L1 cache due to a capacity or a conflict miss. But a capacity or a conflict miss will never happen on the first core since the data that is in the private cache will be invalidated as soon as the second core tries to modify the data.

Basically since the shared cache is not inclusive the modified values will be invalidated in the private caches without being written into the shared cache. This will cause the cores to miss each time they try to modify a shared cache line.

## 3.7 Summary

In order to reduce the number of misses due to the coherency protocol we want the threads that are sharing data to be scheduled on the same chip so that they can share their data on the last level cache. The challenge is to automatically detect data sharing between the threads as well as estimating whether data sharing is significant enough that it is worth relocating the threads onto the same cache. Addressing this challenge is the subject of this thesis.

# Chapter 4

# Measuring the Amount of Sharing

This chapter explains the experiments that were run in order to create our model later. We will begin by describing the counters that we studied and our experimental setup. We then describe the benchmarks that we used. We conclude by explaining our model for predicting the benefits of sharing for a multithreaded application.

In order to find a way to measure the amount of sharing we first run some benchmarks that do benefit from sharing a cache. Then we measure the properties of those benchmarks using hardware performance counters, and based on the results we try to figure out a way to measure the amount of data sharing.

## 4.1 Monitoring Coherency Protocol Events

To measure the amount of sharing between two threads we decided to monitor the amount of traffic between the caches when the threads are running on separate caches. From understanding how the MESI protocol works we know that whenever there is a cache miss, the originating core will send a snoop message to all the other cores to find out if there is a copy of the data in any of the other caches. Each time a *Shared* or an *Exclusive* copy is found the responding core will increase the SNOOP_HIT counter[1]. Each time a *Modified* copy is found in a cache, the responding core will increase the SNOOP_HITM counter.

[1]This counter exists on Intel Xeon architecture.

By measuring SNOOP_HIT and SNOOP_HITM we can get a general understanding of how much applications are sharing data. If two threads of an application are running on separate caches and there are no SNOOP_HIT or SNOOP_HITM responses, it means that the data that either of the cores are requesting or modifying is not found in the other cache. If there are a lot of SNOOP_HIT responses it means that most of the data that is being requested by one of the cores is found in the other core's cache. If there are a lot of SNOOP_HITM responses it means that there is write sharing. Usually write sharing is symmetrical, because for generating a SNOOP_HITM both cores must write to the same address.

Unfortunately AMD systems do not provide information about the events of their coherency protocol. On Intel Xeon architecture such information is available. However after studying the counters and talking with an engineer from Intel we concluded that the data that the counters are providing is not valid and is useless. The Intel engineer advised us to use a Nehalem server as this problem is fixed on that architecture. Intel Nehalem systems provide similar information but with more detail. One of the main differences of Intel Nehalem architecture with the older architectures is that it supports the NUMA[2]. In the NUMA architecture memory is divided into a few banks, each bank is assigned to a chip and is considered that chip's local memory. Memory access latency to local memory is smaller than the access latency to a remote memory bank. Accesses to a remote bank are done via interconnects between the chips. The NUMA architecture allows the memory size and the number of cores to grow. NUMA effects add to the complexity of studying data sharing. Therefore we disabled NUMA when we ran our experiments.

There are two types of counters on Nehalem systems:

- **Regular counters:** This counters provide information per core. Each core has its own set of counters. L1 cache misses, number of instructions retired, number of branches decoded, etc are some examples of regular counters.
- **Uncore counters:** Uncore counters provide information about the chip as a whole. The events that these counters count might have been triggered by any of the cores of that chip. L3 cache misses (the last level shared cache), number of global queue

[2]Non-Uniform Memory Access

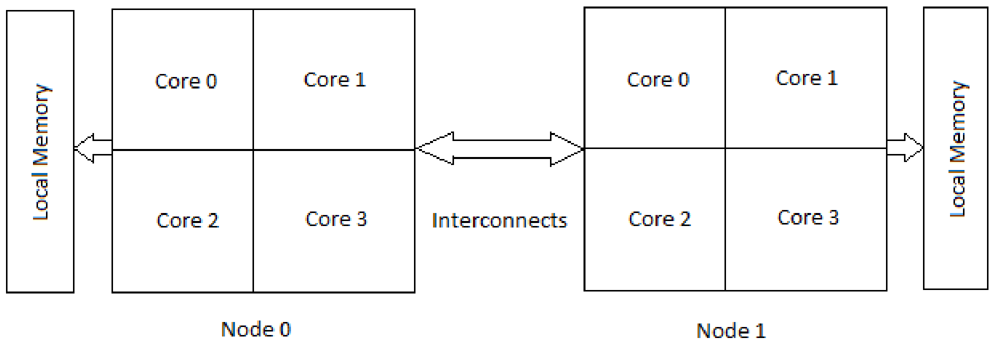

Figure 4.1: NUMA - two node example

allocations, interconnect traffic monitoring counters, etc are some examples of uncore counters.

Counters of interest for measuring sharing are:

- **UNC_SNP_RESP_TO_LOCAL_HOME:** This counter counts the number of times that cores (of other chips) miss in their private L1 and L2, as well as the shared last level L3 cache, and the data which that core is requesting is found in the current chip's L3 cache. In addition to that the original copy of the data in the cache is fetched from the local memory bank of the originating chip.

  For example in figure 4.1 if core 0 of node 0 tries to access a memory address that can be found in the local memory bank (memory of node 0) and misses its shared L3 cache (of node 0), a snoop message will be sent to node 1. Node 1 has the data, so it will trigger the "snoop response to local home" event[3].

[3]Remember that caches on Intel systems are inclusive. It means that if a data cannot be found in the L3

- **UNC_SNP_RESP_TO_REMOTE_HOME:** The only difference between this counter and the previous one is that this counter only counts the snoop responses that the original copy of the data in the memory is fetched from a remote memory bank.

  Similar to the previous example, if core 0 of node 0 tries to access a memory address that can be found in the remote memory bank (memory of node 1) and misses its caches, it will send a snoop message. If a copy is found in the other node's L3 cache, the other chip will trigger this event.

## 4.2 Experimental Setup

We used an Intel Nehalem system as for experimental machine:

- Two four-core Intel Xeon E5520, total of 8 cores, working at 2.2 GHz.
- Each core has a 32 KB instruction cache and a 32 KB data cache.
- Each core has a private 512 KB L2 cache.
- 8 MB of shared L3 cache between the four cores of each chip.
- 12 GB of memory.

Intel Nehalem architecture supports NUMA. Again, in this setup we have disabled NUMA by interleaving the memory.

## 4.3 Experimental Plan

To find out how sharing a cache can help different threads of a multithreaded application first we need to find out applications that would actually share data between different threads. This was not an easy task. Programmers usually try to avoid sharing any data between the threads because they want to minimize the traffic between the cores. Therefore, most of the known applications are designed so that the threads share minimum data in order for the benchmark to be scalable. We could not find any well known benchmark that could benefit

cache, it cannot be found in L1 and L2 caches of that chip. Compared to AMD systems this greatly reduces the number of snoops required to find out if there is a copy.

from sharing a last level cache.

In order to study the effects of data sharing between the threads we wrote several simple benchmarks that made it easy to isolate and understand this effect (some are real applications or just slight modifications to real applications):

- **SwapTest:** This benchmark runs with two threads and it is highly configurable. Upon start up a main thread initializes an integer array of predefined size. For each item in the array a spin lock is initialized. Then the main thread creates two threads and passes the integer array and the array of locks to the two threads.

  The task of each thread is to randomly select two items in the integer array and swap them. In order to swap the items the thread first needs to lock the two items and then swap the numbers. In order to avoid deadlocks each thread locks the number with the lower index first and then locks the other item. The number of swaps is predefined. Each thread is bound to a specific core.

  The two threads are sharing the integer array and the array of locks. When a thread (remember that each thread is bound to a core) modifies its data by locking and modifying an item, it causes the data in the other core's cache to get invalidated. The next time the other core needs to read the data it will miss its cache. This benchmark is designed carefully so that the threads affect each other only by changing the shared arrays. Meaning that there are no hidden cache misses due to false sharing or due to unsafe C system function calls.

  One needs to be careful while setting the size of the integer array. If the size of the array is so small that the whole array fits into a single cache line, each access to the array will result in a cache miss. If the size of the array is so big that it will not fit in the last level cache, then the misses will not be only coherency misses, there will be some capacity misses as well. In the latter case the number of coherency misses will decrease greatly because the chances of the two threads choosing the same item in the array decreases.

- **BucketSortTest:** This benchmark implements a bucket sort. On start up a main thread initializes an array of integers and passes it to two threads. Each thread will run the bucket sort algorithm on a portion of the shared array. In order to increase the value of the bucket counter the threads need to lock a mutex.

- **NetworkTest:** NetworkTest is a benchmark designed to test the effects of data sharing in the internal operating system buffers. This benchmark runs a server and a client thread. The server thread listens to an incoming port. When it receives some data on the incoming port it will write the same data back to the network socket.

  The client thread connects to the server thread and sends a string of predefined size to the server. The number of times the message is sent to the server can be configured as one of the parameters to this benchmark.

- **MySqlTest:** This benchmark is designed to measure the effects of data sharing in a MySql server. The benchmark creates a table with four columns (first column is a string, the rest are numbers) and two indices. Then two threads are spawned.

  Each thread inserts some random values to the table that the main thread creates. MySql server needs to update the indices based on the data that each thread inserts into the table.

- **Memcached:** Memcached is a memory caching system for databases. It uses a hash table and can distribute the data across multiple machines. This application is used by well-knows sites such as YouTube, Reddit, Facebook, and Twitter. Memcached server accepts messages from the network. Each client can ask the server to store some data. In order to use this benchmark we modified the memcached server so that instead of waiting for the clients to send some data, it will generate some data itself. In this way we by pass the need for the clients and the effects of internal operating system buffers.

- **LinearRegression:** LinearRegression was taken from the examples of the Phoenix MapReduce API developed in Stanford University. This sample application does a linear regression analysis on a bunch of random generated points. Pthread-ed version of the LinearRegression example is suffering from false sharing due to poor programming.

This is a real world example of applications that operating system can help improve the performance without the need to modify the source code. In this case, of course, the source code is available and improvements will be much more if a programmer could fix the problem of false sharing.

- **SpecJBB:** This benchmark is designed by SPEC to measure the performance of server-sid Java. This benchmark "emulates a 3-tier system which is the most common type of server-side Java applications"[4]. This benchmark is designed so that the threads do not share much data and therefore cannot benefit from sharing the last level cache.

In the first set of experiments we wanted to see if these benchmarks could benefit from a shared last level cache. To test this we ran each benchmark with two different configurations. In the first configuration the threads of each benchmark are bound to the cores that share a last level cache, and in the other configuration the threads are bound to cores in separate chips.

### 4.3.1 SwapTest Results

Table 4.1 compares the execution time of the SwapTest benchmark when the threads were bound to a single chip (thus a shared last level cache) versus when the threads were bound to cores on separate chips.

The last column of 4.1 is the ratio of the runtime on a shared cache to the runtime on separate chips. Lower ratio means that the benchmark can benefit more from being placed on a shared cache. As the table shows the benefits of running on a shared last level cache decreases as the shared data size increases. This is because when the shared data size does not fit the last level cache most of the misses will be due to limited cache size. It is unlikely that the cores modify the same item. Therefore the number of coherency misses will dwindle and the benchmark will not benefit from running on the shared cache. This is shown visually in figure 4.2.

[4]From the homepage of SpecJBB

Table 4.1: SwapTest running on a shared last level cache versus running on separate chips. Lower ratio means a higher benefit from being placed on a shared cache.

| Array Size | Runtime on a Shared Cache | Runtime on Separate Chips | Ratio |
|---|---|---|---|
| 10 | 7.18 | 11.96 | 0.60 |
| 25 | 7.08 | 12.20 | 0.58 |
| 100 | 6.02 | 9.61 | 0.63 |
| 400 | 5.18 | 7.18 | 0.72 |
| 800 | 4.49 | 5.84 | 0.77 |
| 2,000 | 4.98 | 6.60 | 0.75 |
| 4,000 | 4.36 | 5.50 | 0.79 |
| 8,000 | 3.98 | 4.78 | 0.83 |
| 16,000 | 3.72 | 4.22 | 0.88 |
| 40,000 | 4.17 | 4.85 | 0.86 |
| 100K | 4.68 | 5.70 | 0.82 |
| 200K | 4.60 | 5.40 | 0.85 |
| 400K | 4.77 | 5.28 | 0.90 |
| 800K | 5.10 | 5.48 | 0.93 |
| 1M | 6.03 | 6.69 | 0.90 |
| 2M | 10.09 | 10.16 | 0.99 |
| 4M | 12.28 | 12.26 | 1.00 |
| 8M | 13.65 | 13.51 | 1.01 |
| 16M | 15.69 | 16.03 | 0.98 |

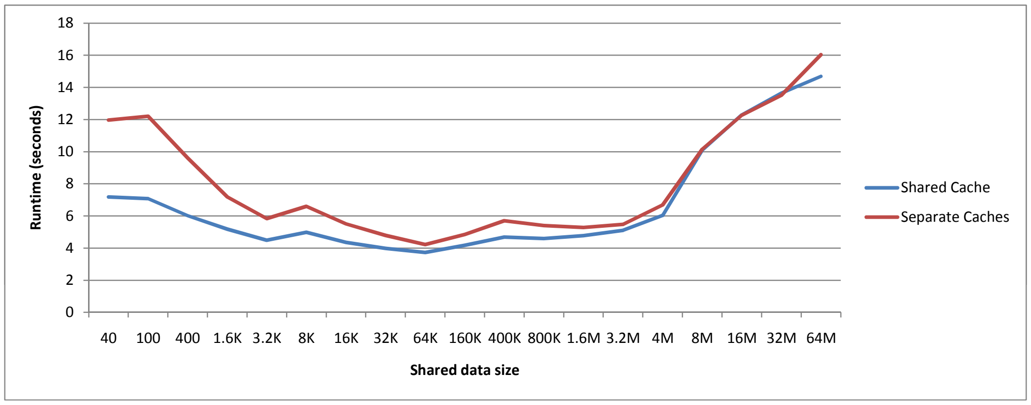

Figure 4.2: SwapTest Runtime on a Shared Cache and on Separate Caches

### 4.3.2 BucketSort Results

Figure 4.3 shows the execution time of the BucketSort benchmark when it is running on a shared last level cache versus when it is running on separate chips. As the results show, regardless of the parameters this benchmark always benefit from running on a shared last level cache.

### 4.3.3 NetworkTest

Table 4.2: Execution time of Network test on a shared last level cache versus separate caches

| Buffer Size | Runtime on a Shared Cache | Runtime on Separate Chips | Ratio |
|---|---|---|---|
| 25 Bytes | 15.49 | 23.75 | 0.65 |

Table 4.2 shows the execution time of NetworkTest when running on a shared last level cache versus running on separate chips.

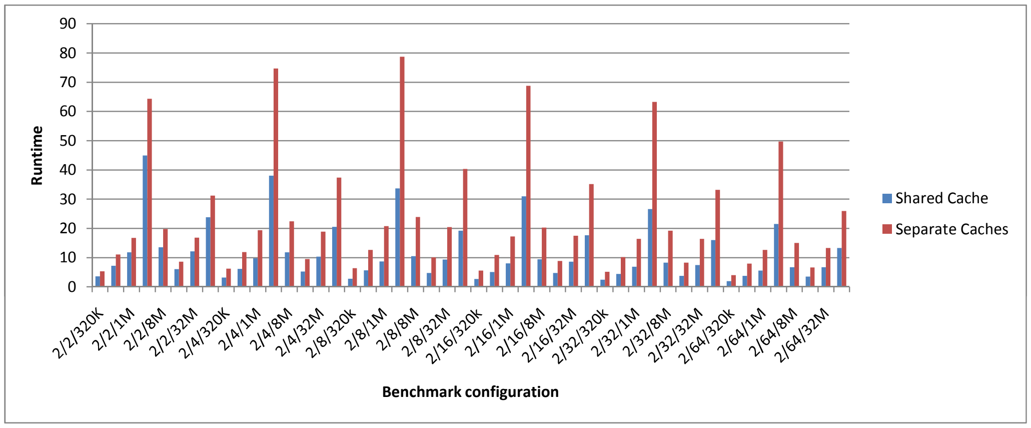

Figure 4.3: BucketSortTest Runtime on a Shared cache and on Separate Caches

Table 4.3: Execution time of MySqlTest on a shared last level cache versus separate caches

| Number of rows | Runtime on a Shared Cache | Runtime on Separate Chips | Ratio |
|---|---|---|---|
| 1M | 10.59 | 14.83 | 0.71 |

### 4.3.4 MySqlTest

Table 4.3 shows the execution time of MySqlTest when running on a shared last level cache versus running on separate chips.

### 4.3.5 Looking at Hardware Performance Counters

The execution times for the four benchmarks that we ran shows potential benefits of scheduling threads that share data on cores that share the last level cache. We need to find a dynamic way for detecting that threads are sharing data and report proper information to the operating system to help it make scheduling decisions. As explained earlier some hardware performance counters seem promising for this goal.

Table 4.4 shows the preliminary results for SwapTest when it is being run on **separate chips**. *L3_MISSES* is the number of last level cache misses, this includes the misses due to the coherency protocol. *Total uncore sharing snoops* is UNC_SNP_RESP_TO_LOCAL _HOME + UNC_SNP_RESP_TO_REMOTE_HOME. It shows the total number of snoops that were hit in the other cache.

As the results show when the size of the shared data increases the number of hit snoops

Table 4.4: Snoops and L3 misses for SwapTest

| Array Size | Total uncore sharing snoops | L3_MISSES |
|---:|---:|---:|
| 10 (40 bytes) | 160.17M | 196.99M |
| 25 (100 bytes) | 261.28M | 284.44M |
| 100 (400 bytes) | 245.52M | 255.17M |
| 400 (1.6 KB) | 189.90M | 192.31M |
| 800 (3.2 KB) | 122.24M | 123.40M |
| 2,000 (8 KB) | 180.76M | 181.24M |
| 4,000 (16 KB) | 122.66M | 122.89M |
| 8,000 (32 KB) | 68.38M | 68.57M |
| 16,000 (64 KB) | 40.14M | 40.22M |
| 40,000 (160 KB) | 71.54M | 71.61M |
| 100K (400 KB) | 117.73M | 117.90M |
| 200K (800 KB) | 73.00M | 73.20M |
| 400K (1.6 MB) | 41.14M | 41.43M |
| 800K (3.2 MB) | 22.17M | 27.53M |
| 1M (4 MB) | 71.30M | 94.67M |
| 2M (8 MB) | 32.16M | 202.25M |
| 4M (16 MB) | 15.08M | 284.28M |
| 8M (32 MB) | 9.01M | 324.81M |
| 16M (64 MB) | 7.55M | 352.61M |

decrease. We have already explained why this happens. Last level cache misses however, have an interesting trend. They go down as we increase the data size, then they go up again as the data size goes beyond the last level cache size. When the data size is small almost every access results in an invalidation and thus a miss in the other cache. This happens because all data can be fit into a few cache lines. Each core chooses a number (which is in one of the cache lines) randomly. Therefore the chances that the two cores choose the same cache line is high. When the data size increases, the number of cache lines that the cores access increases as well. Therefore the chances of picking the same cache line reduces and not every access results in an invalidation, which decrease the number of cache misses. As the data size goes beyond the cache size, capacity misses start to dominate the misses.

Table 4.5 shows how we analyze this data. To understand this table we need to understand what each column means:

- **Runtime Improvement:** Ratio of execution time when threads are bound so that they share a last level cache to execution time when threads are not sharing a last level cache. Lower ratio means better improvement from co-locating on a shared cache.
- **SMR Separate:** Ratio of the total number of snoops to the total number of last level cache misses when threads are not sharing a last level cache($\frac{\#Snoops}{\#Misses}$ Separate)
- **MPmI Shared:** Cache misses per million instructions when running on a shared cache configuration.
- **MPmI Separate:** Cache misses per million instructions when running on a separate cache configuration.

Looking closely at the values for *SMR Separate* we notice that when the array size is small the value for *SMR Separate* is a little below one. This is unexpected according to our understanding of the caches and the coherency protocol. We do not know the reason behind this, one guess is that the processor is doing some kind of an optimization when it notices the high coherency miss on a single cache line.

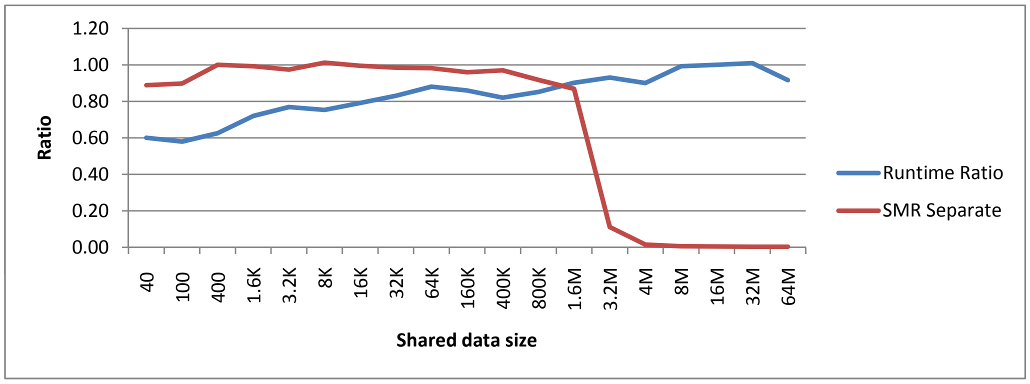

Figure 4.4: SwapTest Analysis - Runtime Ratio and SMR Separate

Table 4.5: SwapTest Snoop and Cache Miss Analysis

| Array Size | Runtime Ratio | SMR Separate | MPmI Shared | MPmI Separate |
|---:|:---|:---|---:|---:|
| 10 | 0.60 | 0.81 | 44.39 | 12777.78 |
| 25 | 0.58 | 0.92 | 42.39 | 18918.50 |
| 100 | 0.63 | 0.96 | 28.92 | 17557.87 |
| 400 | 0.72 | 0.99 | 30.71 | 13364.19 |
| 800 | 0.77 | 0.99 | 27.08 | 8589.03 |
| 2,000 | 0.75 | 1.00 | 25.75 | 12591.11 |
| 4,000 | 0.79 | 1.00 | 25.71 | 8541.74 |
| 8,000 | 0.83 | 1.00 | 21.66 | 4767.78 |
| 16,000 | 0.88 | 1.00 | 22.05 | 2796.47 |
| 40,000 | 0.86 | 1.00 | 27.79 | 4975.74 |
| 100K | 0.82 | 1.00 | 27.04 | 8187.09 |
| 200K | 0.85 | 1.00 | 30.29 | 5080.35 |
| 400K | 0.90 | 0.99 | 29.57 | 2871.82 |
| 800K | 0.93 | 0.81 | 305.90 | 1903.60 |
| 1M | 0.90 | 0.75 | 2345.27 | 6536.19 |
| 2M | 0.99 | 0.16 | 13077.79 | 13874.62 |
| 4M | 1.00 | 0.05 | 19050.76 | 19260.97 |
| 8M | 1.01 | 0.03 | 21545.35 | 21485.22 |
| 16M | 0.98 | 0.02 | 22203.29 | 22246.62 |

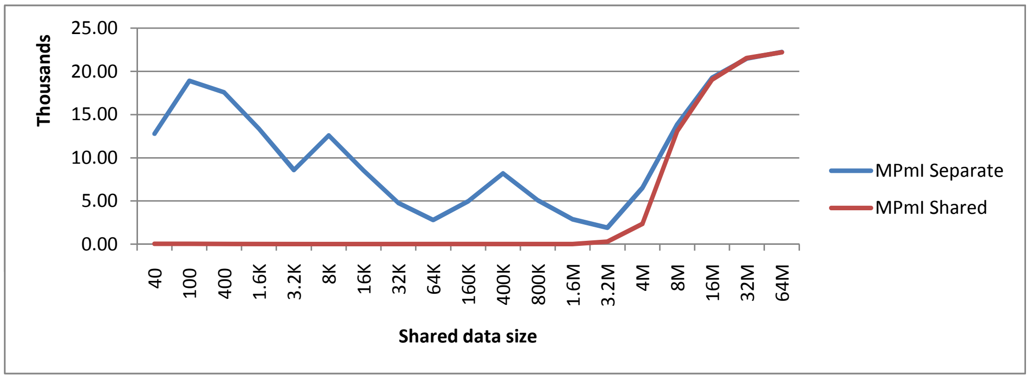

Figure 4.5: SwapTest Analysis - MPmI Separate and MPmI Shared

We know that SwapTest shares data between the threads and can benefit from being scheduled on a shared cache configuration. By looking at figure 4.5 *MPmI Shared* and *MPmI Separate* clearly show that if SwapTest get scheduled on separate chips the number of cache misses for it will increase (and therefore its performance will decrease). As the shared data size increases there is no difference between the number of misses of the two configurations. The runtime ratio shows no improvement as well (as shown in figure 4.4).

The problem with just looking at MPmI is that for an unknown application we cannot differentiate the cache misses due to coherency and the cache misses due to capacity misses. For differentiating these two we need to look at other counters. Here is where *SMR Separate* comes in handy. As the data shows *SMR Separate* is close to one when the data fits the cache. When the data gets bigger *SMR Separate* decreases until it reaches zero. When the value of *SMR Separate* is close to one it means that for most of the misses there has been a snoop. When it is close to zero it means that misses are happening without a snoop hit, therefore we can safely assume that this kind of misses are capacity misses.

Tables 4.8, 4.6, and 4.7 show the same analysis for BucketSort, NetworkTest and MySqlTest respectively.

Table 4.6: NetworkTest Snoop and Cache Miss Analysis

| Buffer Size | Runtime Ratio | SMR Separate | MPmI Shared | MPmI Separate |
|---:|---|---|---:|---:|
| 25 | 0.65 | 0.98 | 14.56 | 6715.25 |

Table 4.7: MySqlTest Snoop and Cache Miss Analysis

| Number of Rows | Runtime Ratoi | SMR Separate | MPmI Shared | MPmI Separate |
|---:|---|---|---:|---:|
| 1M | 0.71 | 0.90 | 325.42 | 2511.96 |

## 4.4 Summary

All the values in the tables can be obtained online. Now we understand that *SMR Separate* and *MPmI Separate* can somehow be used to identify data sharing. In the next chapter we explain how we can use these two to measure the amount of data sharing and to predict the benefits of co-locating threads on a shared cache.

Table 4.8: BucketSort Snoop and Cache Miss Analysis

| Parameters | Runtime Ratio | SMR Separate | MPmI Shared | MPmI Separate |
|---|---|---|---|---|
| 2/2/320K | 0.68 | 0.70 | 122.82 | 12839.89 |
| 2/2/640K | 0.65 | 0.71 | 94.44 | 12263.37 |
| 2/2/1M | 0.71 | 0.68 | 93.08 | 11441.84 |
| 2/2/4M | 0.70 | 0.66 | 894.31 | 13860.12 |
| 2/2/8M | 0.69 | 0.66 | 862.53 | 13798.97 |
| 2/2/16M | 0.70 | 0.61 | 884.43 | 10424.24 |
| 2/2/32M | 0.72 | 0.62 | 1043.45 | 11011.04 |
| 2/2/64M | 0.76 | 0.61 | 875.13 | 10682.40 |
| 2/4/320K | 0.52 | 0.81 | 87.51 | 16183.83 |
| 2/4/640k | 0.52 | 0.80 | 92.58 | 16520.24 |
| 2/4/1M | 0.50 | 0.81 | 90.44 | 15666.56 |
| 2/4/4M | 0.51 | 0.78 | 871.22 | 18488.75 |
| 2/4/8M | 0.53 | 0.76 | 891.89 | 17428.12 |
| 2/4/16M | 0.55 | 0.74 | 930.21 | 14060.15 |
| 2/4/32M | 0.55 | 0.73 | 911.39 | 13972.09 |
| 2/4/64M | 0.55 | 0.73 | 893.38 | 13956.24 |
| 2/8/320k | 0.44 | 0.90 | 105.93 | 22297.98 |
| 2/8/640k | 0.45 | 0.90 | 71.43 | 22195.36 |
| 2/8/1M | 0.42 | 0.90 | 69.95 | 22303.48 |
| 2/8/4M | 0.43 | 0.87 | 879.87 | 23769.65 |
| 2/8/8M | 0.44 | 0.87 | 898.21 | 22341.02 |
| 2/8/16M | 0.47 | 0.85 | 908.58 | 17737.25 |
| 2/8/32M | 0.46 | 0.84 | 904.72 | 17739.00 |
| 2/8/64M | 0.48 | 0.83 | 894.97 | 17589.47 |
| 2/16/320k | 0.48 | 0.92 | 76.15 | 20021.04 |
| 2/16/640k | 0.47 | 0.91 | 74.49 | 20016.00 |
| 2/16/1M | 0.47 | 0.92 | 77.31 | 20220.40 |
| 2/16/4M | 0.45 | 0.89 | 886.63 | 21874.99 |
| 2/16/8M | 0.46 | 0.88 | 902.29 | 20695.63 |
| 2/16/16M | 0.53 | 0.86 | 905.40 | 16086.24 |
| 2/16/32M | 0.49 | 0.84 | 1087.48 | 16362.63 |
| 2/16/64M | 0.50 | 0.84 | 895.15 | 16345.60 |
| 2/32/320k | 0.48 | 0.94 | 74.52 | 22416.70 |
| 2/32/640k | 0.44 | 0.94 | 69.43 | 22477.71 |
| 2/32/1M | 0.42 | 0.94 | 65.21 | 22516.82 |
| 2/32/4M | 0.42 | 0.91 | 885.06 | 24295.00 |
| 2/32/8M | 0.43 | 0.90 | 902.45 | 22809.68 |
| 2/32/16M | 0.46 | 0.89 | 919.60 | 17919.49 |
| 2/32/32M | 0.45 | 0.89 | 909.93 | 17952.25 |
| 2/32/64M | 0.48 | 0.89 | 896.39 | 17965.93 |

# Chapter 5

# Analysis and Implementation

In this chapter we analyze the data we have gathered and build a model that will help us predict to what extent threads will benefit from sharing-aware placement. We explain what tools we used to make our sharing aware scheduler and how it works.

## 5.1 Model

From the results of the previous section we already know that we can detect whether two threads are sharing any data or not by looking at *SMR Separate* and *MPmI Separate*. In this section we introduce a way to predict how much performance will improve if we migrate the threads to share the last level cache.

Table 4.5 shows the performance improvements of SwapTest along with the analysis of the snoops and the number of last level cache misses. From that table we understand that the bigger the *SMR Separate* the more the threads are sharing data and the more the performance will improve if we migrate the threads to a shared cache. Looking at *SMR Separate* alone however, is not enough.

Consider a case where an application does not have a lot of cache misses. The only few cache misses that it generates are because of sharing. In this case *SMR Separate* will be close to one since all the misses are due to sharing. But the application cannot benefit from a shared last level cache, because it is not generating a lot of cache misses and resolving those few misses will not have a great impact on the performance of the application. Due to this problem we need to look at both *SMR Separate* and *MPmI Separate*.

When *SMR Separate* is close to one and there are a lot of cache misses (*MPmI Separate* is large) it means that the application is generating a lot of misses and those misses are coherency ones. So the performance of the application will improve if the scheduler migrates the threads into a shared cache.

If there are a lot of cache misses but *SMR Separate* is close to zero, it means that the cache misses are not due to coherency invalidations. In this case if the scheduler migrates the threads into a shared cache the application might suffer. If the misses are due to capacity, by migrating the threads into a shared cache the scheduler is forcing the threads to share a cache, which will result in less capacity for each thread, and therefore an increased number of capacity misses.

We decided to do a linear regression analysis on *SMR Separate*, *MPmI Separate*, and the execution time. To do the analysis we needed a complete data set. We need to run the linear regression analysis to get the coefficients for our model:

$$\textit{Runtime Ratio} = x * (SMR\ Separate) + y * (MPmI\ Separate) + C$$

We already had the data for the correlation of *SMR Separate* and *MPmI Separate* when the performance improves. For a complete data set we needed some data for when the performance does not improve.

To gather this data we modified our benchmarks to do the exact same operation on their data, but the data would not be shared between the threads. The array was shared between the threads by the main thread. The main thread would allocate an array and would pass it to the two threads. After the modification, the main thread allocates two arrays and passes each array to one thread. In this way the threads will do the exact same thing, but on different data. Obviously in this case the benchmarks could not benefit from sharing.

Table 5.1 shows the results of SwapTest after we modified it. Note that for small array sizes *SMR Separate* is close to one, but there is no significant performance improvements because *MPmI Separate* is small. Also note that how *SMR Separate* will go to zero as *MPmI Separate* increases. The small values for *MPmI Separate* when the data size is small means that the threads are not invalidating the data in other caches (compare these values to those of table 4.5).

Table 5.1: SwapTest-No Data Sharing Snoop and Cache Miss Analysis

| Array Size | Runtime Improvement | SMR Separate | MPmI Shared | MPmI Separate |
|---:|---|---|---:|---:|
| 10 | 0.95 | 0.92 | 28.70 | 26.96 |
| 25 | 0.95 | 0.85 | 28.17 | 29.80 |
| 100 | 0.96 | 0.79 | 28.50 | 29.09 |
| 400 | 0.95 | 0.97 | 27.21 | 310.04 |
| 800 | 0.97 | 0.82 | 26.23 | 27.83 |
| 2,000 | 0.95 | 0.97 | 23.03 | 25.37 |
| 4,000 | 0.97 | 0.96 | 23.18 | 23.90 |
| 8,000 | 0.99 | 0.83 | 21.89 | 36.40 |
| 16,000 | 0.99 | 0.96 | 23.12 | 27.37 |
| 40,000 | 0.97 | 0.97 | 23.57 | 24.06 |
| 100K | 0.96 | 0.91 | 27.60 | 31.21 |
| 200K | 0.97 | 0.88 | 27.38 | 25.59 |
| 400K | 0.97 | 0.73 | 180.40 | 36.12 |
| 800K | 0.98 | 0.04 | 922.31 | 888.78 |
| 1M | 1.06 | 0.02 | 6921.20 | 2608.85 |
| 2M | 1.01 | 0.00 | 14494.20 | 13120.57 |
| 4M | 1.01 | 0.00 | 19502.31 | 19189.99 |
| 8M | 1.01 | 0.00 | 22004.41 | 21461.18 |
| 16M | 1.01 | 0.00 | 22854.76 | 22606.72 |

Similar results were obtained for BucketSort.

## 5.2 Linear Regression

Data was gathered for sharing and non-sharing SwapTest and BucketSort. They were run with different parameters. Table 5.2, 5.3, and 5.4 show the results of linear regression analysis.

Table 5.2: Regression Statistics

| Regression Statistic | |
|---|---|
| Multiple R | 0.90 |
| R Square | 0.81 |
| Adjusted R Square | 0.81 |
| Standard Error | 0.10 |
| Observations | 104 |

Table 5.3: Analysis of Variance

| | df | SS | MS | F | Significance F |
|---|---|---|---|---|---|
| Regression | 2 | 4.58 | 2.29 | 215.80 | 3.43E-37 |
| Residual | 101 | 1.07 | 0.01 | | |
| Total | 103 | 5.65 | | | |

Table 5.4: Linear Regression Coefficients

| | Coefficients | Standard Error |
|---|---|---|
| Intercept | 1.044417 | 0.01784 |
| *MPmI Separate* | -0.000013 | 0.00000 |
| *SMR Separate* | -0.296878 | 0.03065 |

Now that we have the coefficients of the variables (table 5.4) we can create our model that predicts the performance improvements of a migration. We used the data from SwapTest and BucketSort to create our model. Now we can test our model on the rest of the benchmarks: NetworkTest, MySqlTest, and memcached, LinearRegression, and SpecJBB.

Table 5.5 shows the hardware performance analysis of memcached. We measured the runtime improvement by comparing the average service time for each request when running on a shared cache versus running on separate chips.

Table 5.5: memcached Snoop and Cache Miss Analysis

| MemSize | Runtime Ratio | SMR Separate | MPmI Shared | MPmI Separate |
|---|---|---|---|---|
| 1 MB | 0.70 | 1.00 | 20.51 | 3105.32 |
| 2 MB | 0.67 | 1.00 | 19.91 | 2973.12 |
| 4 MB | 0.69 | 1.00 | 20.40 | 3132.60 |
| 8 MB | 0.66 | 1.00 | 19.36 | 3016.21 |
| 16 MB | 0.74 | 1.00 | 21.21 | 3040.51 |
| 32 MB | 0.72 | 1.00 | 20.67 | 2862.99 |
| 64 MB | 0.69 | 1.00 | 20.97 | 2859.36 |

Table 5.6 compares the prediction of our model with the real runtime improvements of memcached.

Table 5.6: Real runtime improvements versus predicted runtime improvements

| Real Runtime Ratio | Predicted Runtime Ratio |
|---|---|
| 0.70 | 0.71 |
| 0.67 | 0.71 |
| 0.69 | 0.71 |
| 0.66 | 0.71 |
| 0.74 | 0.71 |
| 0.72 | 0.71 |
| 0.69 | 0.71 |

Table 5.7 shows how our model works for predicting the performance of NetworkTest, MySqlTest, LinearRegression, and SpecJBB.

Table 5.7: Real runtime improvements versus predicted runtime improvements for NetworkTest, MySqlTest, LinearRegression, and SpecJBB

| Benchmark | Real Runtime Ratio | Predicted Runtime Ratio |
|---|---:|---:|
| NetworkTest | 0.65 | 0.67 |
| MySqlTest | 0.71 | 0.74 |
| LinearRegression | 0.63 | 0.55 |
| SpecJBB | 1.00 | 0.96 |

Figure 5.1 summarizes the results that we got from trying out model.

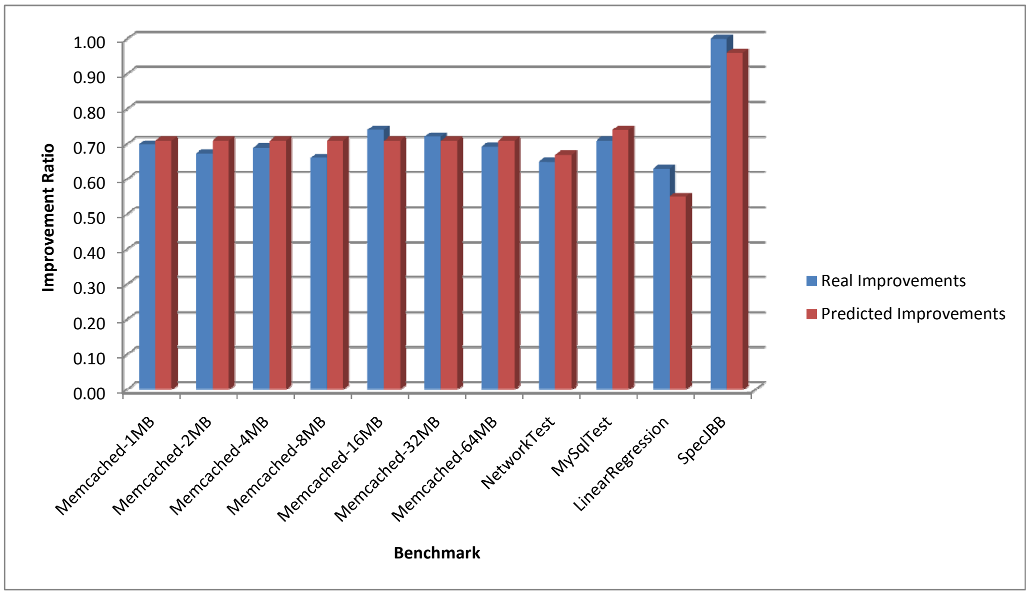

Figure 5.1: Real Improvements vs Predicted Improvements Using Our Model

Some configurations of *BucketSort-NoSharing* not only do not benefit from running on a shared cache, but they actually suffer from it. This is because the threads of this benchmark are competing for the limited shared resource. They can run faster alone if they are given the whole cache. As figure 5.2 shows, when BucketSort-NoSharing runs on a shared cache, the real runtime ratio is larger than 1. Meaning that the benchmark run slower then it was coscheduled on a shared cache. Our model predicts the runtime ratio to be slightly over 1. It means that our model can still work correctly for the applications that suffer from a shared cache. Our model predicts no performance improvements from running on shared cache (actually slightly over 1 means that the model predicts a little decrease in performance!), and therefore the threads must not be scheduled together. An agnostic scheduler might schedule the threads together which will cause the application to suffer.

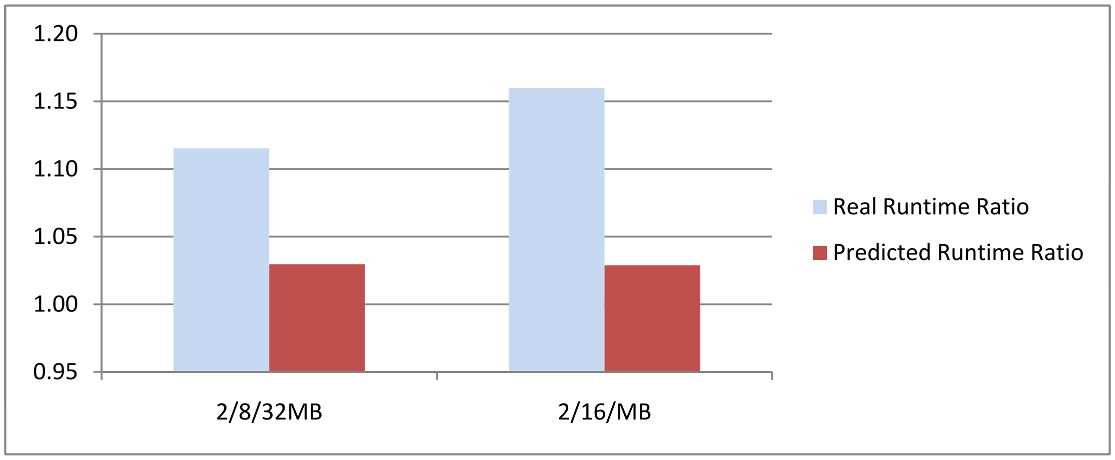

Figure 5.2: Real and Predicted Runtime Ratio for BucketSort-NoSharing

## 5.3 Designing the Scheduler

We implemented our scheduler using the model that we made with the coefficients of the linear regression analysis. The general flow of the scheduler is pretty simple. The scheduler needs to constantly monitor the running threads and measure the performance counters. Having *MPmI Separate* and *SMR Separate* the scheduler can decide whether it makes sense to migrate the threads on to a shared cache or not.

We implemented our scheduler using the perfmon library. The perfmon library helps the developers program the hardware performance counters of CPUs. For measuring the uncore events on Intel Nehalem architecture the scheduler needs to create a system wide monitoring session, and for monitoring the per-CPU events it needs to create a per-thread monitoring session. With a system wide monitoring session the scheduler can gather data about a whole chip while with per-thread monitoring session the scheduler can gather detailed information about each thread that is running on a core. Gathering information per thread is preferred, but due to the limitation of the hardware performance counters some events cannot be measured per thread.

The scheduler starts by attaching to all the threads, call the proper perfmon functions to load the monitoring contexts, and then it starts gathering data. To start a system wide monitoring session the scheduler needs to pause all running threads, unload all per-thread monitoring contexts, and load a system wide context. This is done every second. The following shows the general flow of the scheduler:

```
loop
    Pause_Threads();
    if Is_System_Wide_Context_Loaded() then
        Unload_System_Wide_Context();
    end if
    Load_PerThread_Contexts();
    Resume_Threads();
    $threads$ ← Monitor_PerThread_Data(); {for one second}
    Pause_Threads();
    Unload_PerThread_Contexts();
    Load_System_Wide_Context();
    Resume_Threads();
    $systemWide$ ← Monitor_Uncore_Events(); {for one second}
    $runtimeRatio$ ← Predict($threads$, $systemWide$);
    if $runtimeRatio \leq threshold$ then
        Migrate_Threads();
    end if
end loop
```

## 5.4 Testing the Scheduler

The scheduler was implemented on our Intel Nehalem server. We compare our scheduler with the default scheduler of Linux. Each benchmark was run three times. Figure 5.3 shows the runtime improvement of the benchmarks running with our sharing aware scheduler versus the default scheduler. We ran the benchmarks one at the time. Each benchmark was configured to use only two threads. We could not set the benchmarks to run with more than two threads because our technique cannot determine which threads are sharing if there are more than two threads. This is due to limitation of uncore events. Uncore events are per-chip and do not provide information per thread. Therefore if there are more than one threads running on a chip the scheduler cannot simply figure out which threads are generating those uncore events. We used our Intel Nehalem server to test out scheduler. Our Intel Nehalem server has the following configuration:

- Two four-core Intel Xeon E5520, total of 8 cores, working at 2.2 GHz.
- Each core has a 32 KB instruction cache and a 32 KB data cache.
- Each core has a private 512 KB L2 cache.
- 8 MB of shared L3 cache between the four cores of each chip.
- 12 GB of memory.

As we can see from the figure the scheduler can successfully identify that the benchmarks are sharing and is scheduling the threads on a shared cache. For testing the scheduler we added *Intel.Fibonacci* which is one of the examples of parallel programming using Intel TBB.

The process of monitoring the threads, loading up some contexts, pausing and resuming the threads introduces some overhead. To measure this overhead we ran our SwapTest benchmark and disabled the data sharing (*SwapTest-NoSharing*). In this case the scheduler will constantly monitor the threads, but it finds out that the thread are not sharing data, meaning that the runtime ratio is close to one. Therefore it does not schedule the threads together. When our scheduler does not coschedule the threads on the same chip, it acts like the default scheduler. Both schedulers schedule the threads the same way. However our sharing-aware scheduler does more processing than the default scheduler, and since there are no benefits from our scheduler in this case, all that remains are the negative effects of the extra processing. As we can see in figure 5.3 *SwapTest-NoSharing* suffers from this overhead. However this overhead is about 2% and is negligible.

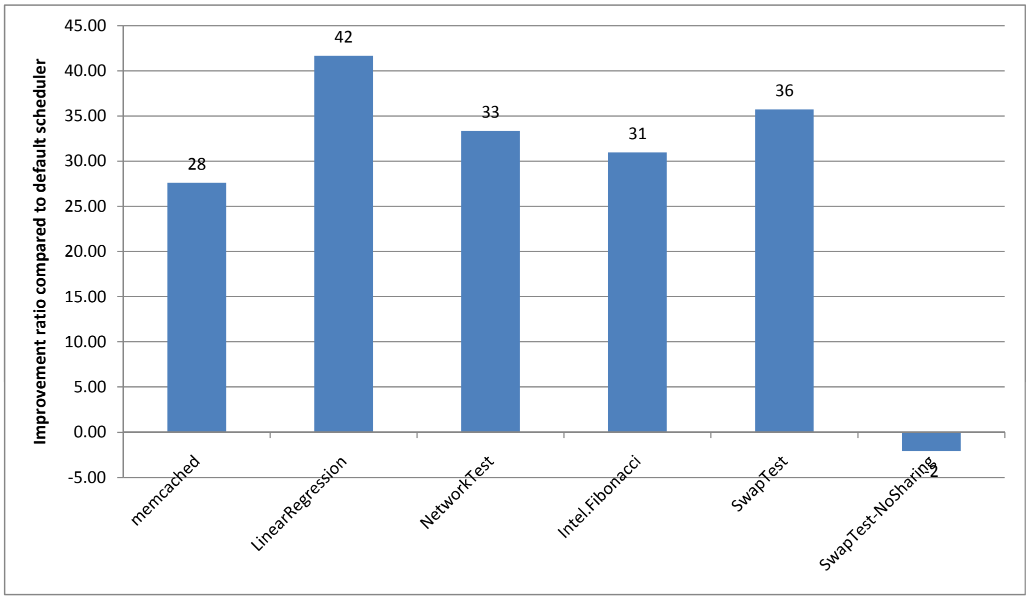

Figure 5.3: Runtime improvements with our sharing aware scheduler compared to the default scheduler

## 5.5 Summary

In this chapter we described how we create our model based on *MPmI Separate* and *SMR Separate*. Using our model we designed a sharing-aware scheduler that can successfully identify sharing and migrate the threads to a shared cache if there is data sharing.

Our scheduler constantly monitors hardware performance counters to find out if there is any data sharing between the threads. If there is data sharing between the threads the scheduler places the threads onto a shared cache to reduce the coherency misses and cross-chip coherency messages. Just by using this technique our scheduler improved the performance of the benchmarks by up to 42%. Our scheduler only uses hardware performance counters, does not rely on the user to provide any information, does not require modifications to user libraries, and it does everything online.

Due to the limitation of the amount of information that the hardware performance counters provide us there are some cases which determining which threads are sharing is not a straightforward task. Since the counters for measuring the number of snoops on each chip give information about the chip as a whole (and not per core), if more that two threads are sharing on the system we cannot figure out which threads are sharing. We can of course detect that there is some data sharing between the threads, but we will not be able to detect which threads are sharing. One solution to this problem might be shuffling the threads on the cores and observing their performance. This can be done by moving threads around the chips and constantly monitoring the performance and the counters. The scheduler can intelligently pick the placing that results in the best performance. In such implementation the scheduler needs to periodically check its decision and monitor the threads again since the applications might have changed their phases or sharing behavior. The overhead of context switching and migrating the threads will be insignificant compared to the benefits that the scheduler will have.

The other limitation of our scheduler is that we need to have the coefficients for the current architecture. We did not have access to different architectures to study whether our coefficients would work well on all architectures.

# Chapter 6

# Conclusion and Future Work

Scheduling applications on multicore systems is a challenging task. In this thesis we provided a way to solve one of the many challenges that operating system schedulers face when it comes to optimal scheduling. The default schedulers of Linux and Solaris are sharing agnostic and therefore cannot schedule the threads so that they could benefit from data sharing. Using the hardware performance counters, we provided a model to not only detect data sharing between threads, but to predict the runtime improvements of scheduling the threads that share data on a shared cache configuration.

To detect the data sharing we used the performance counters relating to cache coherency protocol and observed that applications that share data produce a lot of snoop hits. To create our model we also monitored number of last level cache misses and found out that there is a relation between the ratio of snoops to misses, the total number of last level cache misses per million instructions, and the runtime of an application. We designed a model that could be used in the scheduler to detect among threads and co-locate threads that share data on the same chip. Our scheduler uses perfmon library to monitor the behavior of the threads in the system. It uses our model to determine whether some threads will benefit from being scheduled on a shared cache configuration, and will schedule them in that manner if the model predicts runtime improvements. Using our scheduler we were able to achieve runtime improvements of up to 42%. This potential improvements come with our scheduler only for applications that the amount of data sharing is significant. For the applications that do not share data, our sharing-aware scheduler acts like the default scheduler, but has an overhead of 2%.

# Bibliography

[1] Frank Bellosa and Martin Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 37:113–121, August 1996.

[2] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. *High-Performance Computer Architecture, International Symposium on*, 0:340–351, 2005.

[3] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115, New York, NY, USA, 2007. ACM.

[4] S. J. Eggers and R. H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 257–270, New York, NY, USA, 1989. ACM.

[5] W.-K. Hong, N.-H. Kim, and S.-D. Kim. Design and performance evaluation of an adaptive cache coherence protocol. *Parallel and Distributed Systems, International Conference on*, 0:33, 1998.

[6] Aamer Jaleel, Robert S. Cohn, Chi keung Luk, and Bruce Jacob. Cmp$im: A binary instrumentation approach to modeling memory behavior of workloads on cmps. Technical report, 2006.

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

[8] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. *SIGOPS Oper. Syst. Rev.*, 41(3):47–58, 2007.

[9] R. Thekkath and S. J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *ISCA '94: Proceedings of the 21st annual international symposium on Computer architecture*, pages 176–186, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[10] J. Torrellas, H. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Trans. Comput.*, 43(6):651–663, 1994.

[11] Boris Weissman. Performance counters and state sharing annotations: a unified approach to thread locality. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 127–138, New York, NY, USA, 1998. ACM.

[12] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 203–212, New York, NY, USA, 2010. ACM.

[13] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 129–142, New York, NY, USA, 2010. ACM.