Performance Modeling of Multi-core Systems

Caches and Locks

XIAOYUE PAN
Performance is an important aspect of computer systems since it directly affects user experience. One way to analyze and predict performance is via performance modeling. In recent years, multi-core systems have made processors more powerful while keeping power consumption relatively low. However the complicated design of these systems makes it difficult to analyze performance. This thesis presents performance modeling techniques for cache performance and synchronization cost on multi-core systems.

A cache can be designed in many ways with different configuration parameters including cache size, associativity and replacement policy. Understanding cache performance under different configurations is useful to explore the design choices. We propose a general modeling framework for estimating the cache miss ratio under different cache configurations, based on the reuse distance distribution. On multi-core systems, each core usually has a private cache. Keeping shared data in private caches coherent has an extra cost. We propose three models to estimate this cost, based on information that can be gathered when running the program on a single core.

Locks are widely used as a synchronization primitive in multi-threaded programs on multi-core systems. While they are often necessary for protecting shared data, they also introduce lock contention, which causes performance issues. We present a model to predict how much contention a lock has on multi-core systems, based on information obtainable from profiling a run on a single core. If lock contention is shown to be a performance bottleneck, one of the ways to mitigate it is to use another lock implementation. However, it is costly to investigate if adopting another lock implementation would reduce lock contention since it requires reimplementation and measurement. We present a model for forecasting lock contention with another lock implementation without replacing the current lock implementation.

Keywords: performance modeling, performance analysis, multi-core, cache, lock

Xiaoyue Pan, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Xiaoyue Pan 2016

ISSN 1651-6214
urn:nbn:se:uu:diva-271124 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-271124)
List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I am the primary author and investigator of this paper.

II  Modeling Cache Coherence Misses on Multicores.
I am the primary author and investigator of this paper.

III  Predicting the Cost of Lock Contention in Parallel Applications on Multicores Using Analytic Modeling.
I am the primary author and investigator of this paper. Jonatan Lindén contributed to discussions and implementations.

IV  Forecasting Lock Contention Before Adopting Another Lock Algorithm.
I am the primary author and investigator of this paper. David Klaftenegger contributed to discussions and benchmarks.

Reprints were made with permission from the publishers.
Acknowledgements

First of all, I would like to thank my supervisor Bengt Jonsson for his patient guidance, insightful discussions, and support throughout my PhD. I couldn’t have done this without you, Bengt. Thank you so much. I am very grateful to my co-supervisor Wang Yi for his encouragement, which really helped me through tough times.

It has been a great experience working with my co-authors Jonatan Lindén and David Klaftenegger. Thank you very much for your collaboration, interesting ideas and discussions.

Members of the Real Time Systems group have created a friendly and relaxing work environment, which I greatly appreciate. I would like to thank the current and previous members of the group: Wang Yi, Kai Lampka, Philipp Rümmer, Jonas Flodin, Aleksandar Zeljic, Peter Backeman, Morteza Mohaqeqi, Syed Md Jakaria Abdullah, Nan Guan, Mingsong Lv, Chuanwen Li, Martin Stigge, Pontus Ekberg, Yi Zhang and Pavel Krčál.

I have enjoyed discussions with members of the Computer Architecture team: Nikos Nikoleris, Muneeb Khan, Germán Ceballos, Magnus Själander, Vasileios Spiliopoulos, Erik Hagersten, David Black-Schaffer, Ricardo Alves, Alexandra Jimborean, David Eklöv, Andreas Sandberg, Andreas Sembrant. Thank you all for broadening my research view.

My PhD life would not have been so interesting without my colleagues and friends. I would like to especially thank David Eklöv for his inspirational monologues on research and economy; Andreas Sandberg for his help in adapting to the PhD life and putting up with me in general; Philipp Rümmer for the photography trips and discussions where I get to sharpen my sarcasm skills; Aleksandar Zeljic for sharing ideas from his brilliant mind; Andreas Sembrant for the movie nights and demonstrating the importance of suiting up; Yunyun Zhu for her hilarious dialect imitations; Jonas Flodin for the joke of the week and teaching me Swedish; Peter Backeman for his out-of-the-box questions; Simon Tschirner for the Friday pub and board games; and Haoyu Liu for sharing her research ideas in agriculture and biology fields and for being a great friend. It has been such fun spending time with you guys.

Many thanks go to Magnus Själander and Bengt Jonsson who translated the summary of this thesis into Swedish. Yunyun Zhu designed and drew the cover of this thesis. Her creativity and hard work have been inspiring. Thank you very much, Yunyun!

Finally, I would like to thank my family. I am grateful to my parents Jingxi Pan and Yanmin Sun for their constant support and for teaching me the importance of perseverance and always having a sense of humor. A very special
thank you goes to my fiancé Ricardo Do Souto Fontes Barreira for his unconditional love and support. I am indebted to Chewie for being the happiness generator and his hospitality at the door every single day.

This work is supported by the Swedish Foundation for Strategic Research through the CoDeR-MP project, and by the Swedish Research Council through UPMARC.
## Contents

1 Introduction........................................................................................................ 9
   1.1 Performance modeling............................................................................ 11
   1.2 Research challenges............................................................................. 12
   1.3 Thesis organization............................................................................... 14

2 Caches on multi-core systems................................................................... 15
   2.1 Cache configuration parameters.......................................................... 15
   2.2 Categories of cache misses................................................................... 17
   2.3 Cache performance of programs......................................................... 17
   2.4 Software profiling for estimating cache miss ratios ......................... 18

3 Locks and lock contention......................................................................... 21
   3.1 Lock performance issues...................................................................... 22
   3.2 Performance of some lock implementations........................................ 23

4 Analytic modeling....................................................................................... 27
   4.1 Discrete-time Markov chains (DTMC)................................................ 27
   4.2 Queueing networks............................................................................... 29

5 Estimating cache miss ratios from reuse distance distributions.............. 33
   5.1 A cache modeling framework............................................................. 33
   5.2 Estimating cache coherence misses..................................................... 39
      5.2.1 Characterizing cache coherence misses .................................... 40
      5.2.2 Predicting the number of coherence misses............................... 40

6 Predicting lock contention......................................................................... 44
   6.1 Predicting the lock contention of non-delegation locks..................... 44
      6.1.1 Modeling a program with locks................................................. 45
      6.1.2 Predicting lock contention......................................................... 46
   6.2 Predicting lock contention of delegation locks................................. 48

7 Conclusion..................................................................................................... 50

8 Sammanfattning på svenska......................................................................... 53

References......................................................................................................... 55
Papers

I A Modeling Framework for Reuse Distance-based Estimation of Cache Performance ................................................. 62
  1 Introduction ................................................................................. 63
  2 Preliminaries ............................................................................. 66
  3 Modeling replacement policies: general framework ............... 68
  4 Taking associativity into account ........................................... 71
  5 Modeling different replacement polices using the general framework ................................................................. 72
  6 Evaluation ................................................................................ 78
  7 Related Work .......................................................................... 81
  8 Conclusion .............................................................................. 82

II Modeling Cache Coherence Misses on Multicores ....................... 86
  1 Introduction ............................................................................. 87
  2 Cache miss categorization ................................................... 91
  3 Notations ................................................................................ 92
  4 Modeling cold, capacity, and conflict misses ....................... 93
  5 Modeling cache coherence misses ........................................ 94
  6 Implementation ..................................................................... 102
  7 Evaluation of our models .................................................... 103
  8 Related work ....................................................................... 106
  9 Conclusion .......................................................................... 107

III Predicting the Cost of Lock Contention in Parallel Applications on Multicores using Analytic Modeling ......................... 112
  1 Introduction ............................................................................ 113
  2 Related work ....................................................................... 115
  3 Program structure ................................................................ 116
  4 Queueing networks ........................................................... 117
  5 Evaluation ............................................................................ 121
  6 Conclusions ........................................................................ 129

IV Forecasting Lock Contention Before Adopting Another Lock Algorithm ................................................................. 134
  1 Introduction ............................................................................ 135
  2 Related work ....................................................................... 139
  3 Understanding the queue delegation lock mechanism ............ 140
  4 Predicting reduction in contention ...................................... 143
  5 Profiling to obtain model parameters .................................... 148
  6 Evaluation ............................................................................ 149
  7 A case study ........................................................................ 152
  8 Conclusion ........................................................................... 153
1. Introduction

In recent years, computers have become an inseparable part of our daily life. Like all tools, computers serve the purpose of making people’s lives easier and more enjoyable by fulfilling the needs of their users. The user requirements can be related to the functionality of a system, such as a download program making an alert sound when the download is complete, a music streaming program remembering the history of the play list, a car airbag ejecting during a collision, or a video game allowing the user to save progress and resume later. The requirements can also be related to performance, for example, the download program supporting a minimum speed of 1 Gbit/s, the airbag ejecting within 0.03 seconds after the collision is detected, the music program playing smoothly, the video game having a frame rate of at least 60 frames per second. Such performance requirements can be just as important as the functional ones. Failing to meet them can make the user experience less enjoyable in the best case, and make the program unusable in the worst case.

Since performance is so important, computers have ever since the 1950s become increasingly powerful to help programs run faster and more efficiently. Traditionally, the main method to increase CPU processing speed has been to increase the CPU clock rate. However, the increases in single-core processor clock rate have slowed down considerably in the last decade since they consume too much power nowadays. To overcome the challenge of offering increased performance without increasing the clock rate, chip manufacturers started putting multiple processing units (cores) on a single chip. Figure 1.1 shows an Intel Core i7-5960X multi-core chip with 8 cores: all cores share an L3 cache and common resources such as the memory controller. Another important development is bridging the gap between the fast processor speed and the slow memory access speed, which can differ by two orders of magnitude. One solution to reduce the latency to access data implemented in modern computer architectures is to store some of the data in cache(s) on the chip, which has a lower access latency than the main memory.

With such powerful features in computers to achieve performance, one would think programs should easily reach their maximum performance. Unfortunately, this is far from reality because it is difficult for programs to exploit these features. Let us take the performance of a program on multi-core as an example to show the potential performance issues. Ideally, the execution speed of a program on an $N$-core processor should be $N$ times the execution speed on a single-core processor. This speed-up may not be achievable for several reasons. First, a program typically consists of serial parts, which have to run
sequentially, and parallel parts which can run on multiple cores simultaneously. An observation, known as Amdahl’s law [2], states that the execution speed of a program is limited by the proportion of its serial parts. This is because no matter how many cores there are to speed up the execution of the parallel parts, the serial parts still need to run on one core. Second, even if a program consists of only parallel parts, these parts are often dependent on each other in different ways: they may all need to reach a synchronization point in order to continue executing, one part may require the data calculated by another part, etc. These dependencies cause parallel parts to wait for each other and lose efficiency. Third, the parallel parts may share resources, such as the memory bus and caches. Competing for resources may cause contention, which leads to increased execution time. For example, when multiple cores access the same memory address simultaneously, the memory contention forces some cores to wait.

![An example multi-core architecture: Intel Core i7-5960X with 8 cores](image)

*Figure 1.1. An example multi-core architecture: Intel Core i7-5960X with 8 cores*

Given the importance of performance and the difficulties for a program to reach high performance, programmers need to be able to understand how their programs utilize and exploit the available hardware features. The programmer needs tools to answer questions such as: is the program utilizing all the available cores, or when the program needs to access data, is it available in the cache?

Often the programmer is interested in his/her program’s performance on not only the current platform, but also in how the program would perform when executing on other platforms that are differently configured. One reason to ask such what-if questions is that the program in most cases will be executed by many different users on many different platforms, e.g., with different numbers of cores, different cache sizes, etc. Another reason is to understand how powerful a platform one must acquire in order to achieve a certain performance,
e.g., the cache size required in order for the program not to spend more than a
certain portion of its execution time on memory accesses. A third reason is that
the current systems may be able to reconfigure dynamically at run-time. Run
time systems may determine how many cores to use for specific programs, or
how much cache to allocate for them.

Such what-if questions is difficult to answer only by measurement, but need
methods that can provide insight into the relationship between program per-
formance and platform configuration. Such methods should preferably only
be based on data that can be obtained from the running program with modest
effort.

1.1 Performance modeling

One way to answer what-if questions about performance, such as those men-
tioned in the previous section, is via performance modeling. In performance
modeling, the behavior of the software on some platform is represented by a
model, which describes key features of this behavior at some level of abstrac-
tion. After building a model of the software when running on some (same or
other) platform, we instantiate the model with parameters of the software, usu-
ally obtained via observation or measurement. Such an instantiated model can
be analyzed by existing techniques to predict performance metrics of interest.

As a simple analogy, let us consider a coffee shop with many customers.
The coffee shop has a number of coffee machines since one may not be enough.
The coffee beans in each machine must be refilled when it is empty, which
takes time. The owner of the coffee shop may be interested in questions such
as how the coffee shop should be configured (number of machines, coffee
beans filling time) so that each customer on average waits only for a certain
amount of time (for example, 2 minutes) given that they come to the shop at a
certain rate or how much time each customer waits on average, given a certain
configuration of the coffee shop.

We can model the behavior of the coffee shop with a queueing network,
where the coffee machines are queueing nodes and customers are jobs arriv-
ing at the nodes to receive service. By instantiating the queueing network
model with parameters, such as the arrival rate of customers at the shop and
the number of machines, and analyzing the resulting model, we then extract
performance metrics, such as the average waiting time of a customer. The pre-
vious questions can be answered by changing the parameters of the model and
doing the analysis.
1.2 Research challenges

In this thesis, we address the challenge of developing performance modeling techniques for predicting important performance metrics of a program on a range of platforms. We will develop models that on the one hand can be analyzed to estimate performance for a range of platform configurations, and on the other hand can be constructed by low-cost software profiling on any available platform. We focus on two essential aspects of multi-core performance: cache performance and synchronization cost.

Caches can be designed in many ways with different configuration parameters including cache size, associativity, and replacement policy. Understanding the cache performance under different configurations is useful for exploring the cache design choices by computer architects as well as help programmers understand the cache behavior of their program. On multi-core systems, each core usually has a private cache where the shared data in private caches is kept coherent so that no obsolete data is accessed. There is a cost to maintain coherence when shared data is modified - cache coherence misses. Quantifying such a cost helps programmers understand the cost of sharing data.

Synchronization is necessary for coordinating threads in parallel programs, but it often becomes a performance bottleneck. In this thesis, we focus on studying the performance of one synchronization primitive: locks. While locks protect shared data, they introduce lock contention. Predicting how much contention a lock has brings insight into the cost of synchronization.

The performance of synchronization and the cache system are interdependent on multi-core systems. Synchronization typically involves modification of shared data, which may trigger cache coherence misses. These increased cache misses increase the memory latency of accessing shared data during synchronization, which in turn increases the time spent in synchronization.

Challenge 1: Predicting cache miss ratios under different configurations.

A cache can be designed in many ways with different configuration parameters such as its size and organization structure. Our challenge is to predict the cache miss ratio of a program under different cache sizes, associativities and replacement policies, using information that can be obtained by low-cost profiling. We have chosen to base our prediction on a program’s reuse distance distribution, since it can be obtained at very low cost. In particular, obtaining the reuse distance distribution can be done with significantly lower overhead than alternative inputs, such as the stack distance distribution [18]. While previous works [47] [18] have addressed this problem, they either handle only some specific types of caches or require an expensive-to-obtain input.

The estimated cache miss ratios can help the cache designers evaluate the different design choices and choose the configuration with the best trade off between cost and performance. This can also be used to guide optimization.
For example, if a smaller cache suffices to keep a high performance, it may be beneficial to switch off part of the cache (if possible) to save energy.

We discuss the background of this challenge in Chapter 2 and address the challenge in Paper I, which is summarized in Section 5.1.

**Challenge 2: Predicting cost of data sharing on multi-cores.**

In modern multi-core systems, a memory system with multiple levels of caches is usually adopted: each core has its own private cache and several cores share a shared cache. To keep the shared data in private caches coherent, when one core modifies the shared data, copies of the shared data in other cores’ private caches are *invalidated*. When those copies are later accessed, they must fetch from the shared cache or even the main memory, since they are no longer valid in their cores’ private caches, causing a cache coherence miss. This shows that data sharing comes with a cost of deteriorating cache performance on multi-core systems.

In this challenge, we estimate the number of cache coherence misses of a multi-threaded program on a multi-core system as the number of cores varies, given some information of its behavior on a single core. The result can be used to guide program optimization. For example, if the way a program accesses its shared data is too costly, reducing the amount of shared data or changing the data access pattern may reduce this cost.

We discuss the background of this challenge in Chapter 2 and address the challenge in Paper II, which is summarized in Section 5.2.

**Challenge 3: Predicting lock contention on multi-core systems.**

As a widely used synchronization mechanism, locks are often used to guarantee mutually exclusive access to shared data. While protecting the shared data, locks also cause threads to wait when their attempt to acquire a lock fails. This waiting time (called *lock contention*) increases the time spent for synchronization, and further prevents the whole program from running faster.

The challenge is to predict the lock contention of a program when running on any number of cores given information that can be collected by profiling on one core. The estimated lock contention indicates how much performance may be lost due to synchronization. This can be helpful to identify whether lock contention is a performance bottleneck. It can further guide the program design in efficient ways to access locks.

We discuss the background in Chapter 3 and address the challenge in Paper III, which is summarized in Section 6.1.

**Challenge 4: Predicting lock contention of another lock implementation.**

If lock contention is shown to be a performance bottleneck, one of the ways to mitigate it is to use another lock implementation that may reduce the lock contention. However, it is costly to investigate if adopting another lock implementation would reduce the lock contention since it requires reimplementation
and measurement. More importantly, after putting the effort of implementa-
tion and measurement, it may conclude that the lock contention can not be
reduced, making this effort a waste.

The challenge is to predict the lock contention of adopting another lock
implementation without actually replacing the lock implementation. This re-
sult can be used to predict if another lock can reduce lock contention, which
provides a guideline for optimizing the lock uses.

We discuss the background in Chapter 3 and address the challenge in Pa-
per IV, which is summarized in Section 6.2.

1.3 Thesis organization

This thesis addresses the research challenges concerning both cache perfor-
mance and synchronization cost. It is structured as follows.

Chapter 2 introduces the background of cache performance: cache param-
eters (size, associativity and replacement policy), categories of cache misses
and current techniques to estimate cache miss ratios.

Chapter 3 presents a background on different types of lock implementa-
tions. Lock implementations are divided into two categories depending on
whether a thread can delegate its critical section to another thread. It also
discusses how lock contention can harm performance and potential ways to
mitigate lock contention.

Chapter 4 discusses the analytic modeling techniques used: Markov chains
and queueing networks. It provides basic background for understanding Pa-
per I, Paper III and Paper IV.

In Chapter 5, we address the challenge of predicting cache performance,
i.e., Challenges 1 and 2. Section 5.1 describes our modeling framework, based
on Markov chains, for estimating a program’s cache miss ratio under differ-
ent cache configurations including size, associativity and replacement policy.
Section 5.2 presents three models to estimate the number of cache coherence
misses of a program on a multi-core platform.

In Chapter 6, we present our techniques for predicting synchronization cost,
i.e. Challenges 3 and 4. Our techniques for predicting lock contention are
based on queueing networks. We describe how these models capture the syn-
chronization behavior of parallel programs and how these models can be used
by programmers.

Chapter 7 concludes this thesis and discusses future work.
2. Caches on multi-core systems

While the speed of processors has increased by 10,000 times from 1980 to 2010, the access speed of dynamic random-access memory (DRAM) only increased by less than 10 times during the same period [22]. Nowadays accessing off-chip main memory takes approximately 300 CPU cycles. Modern computer architectures reduce the gap between processor and memory access speed by inserting faster memory such as caches between the processor and main memory. Some data in main memory can also be stored in the cache. When data accessed by a processor is present in the cache, it can be fetched directly from the cache instead of from main memory. This results in a cache hit, which reduces the access latency. The opposite case is called a cache miss. It is desirable to minimize the number of cache misses.

Unfortunately, fast memory is expensive, implying caches cannot be too large. Therefore, modern computer architectures usually adopt a hierarchical cache structure with multiple levels. Lower level caches are smaller and faster while higher level caches are larger and slower. Figure 2.1 shows an example of a hierarchical cache: Intel’s Core i7. It is organized into three levels: each core has its private level 1 cache of size 32KB and level 2 cache of size 256 KB; several cores share a larger level 3 cache of size 2MB.

2.1 Cache configuration parameters

A cache can be designed and configured in many ways which determine how much data it can store, where to put the data, and which data to evict when the cache is fully loaded with data for the program. Cache performance depends on its configuration. In this section, we discuss three essential configuration parameters: size, associativity and replacement policy.
Cache size: the amount of data that the cache can hold. In a modern computer architecture, a small private cache can fit up to a few hundred kilobytes of data while a larger shared cache can fit tens of megabytes.

Cache associativity: data is transferred between main memory and cache in blocks of fixed size, called cache lines (commonly 64 bytes). A key design decision is where cache lines from main memory can be placed in the cache. One possibility, known as fully associative cache, is to put cache lines in any free position in the cache. This solution is flexible but it is expensive to look for a cache line in the whole cache, especially in a large cache. Another possibility, known as directly mapped cache, is to assign a single position for each cache line based on its address. Since the caches are much smaller than the address space of cache lines, multiple cache lines will be assigned to the same position. The consequence of this design is that when a cache line is assigned to a position which is not free, it needs to evict the current cache line. Such an eviction is not necessary if the cache line can be stored in other free positions in the cache.

A third possibility, known as set-associative cache, organizes cache lines into sets. Each cache line is assigned a set based on its address. Within each set, a cache line can be put into any free position. The associativity of a set-associative cache is the number of cache lines in each set. For example, Figure 2.2 shows a cache with associativity 4 where each row shows a cache set consisting of 4 cache lines. The fully associative and the directly mapped caches can be seen as extreme cases of set-associative caches, where a directly mapped cache corresponds to a set size of 1 and a fully associative cache corresponds to the case where the set size equals the cache size.

![A cache set](image)

**Figure 2.2.** A 4-way associative cache

Cache replacement policy: since no cache is infinitely large, an existing cache line must be evicted when a new cache line needs to be stored which does not fit in the current cache. Popular replacement policies include least recently used (LRU), which evicts the least recently used cache line, Pseudo LRU (PLRU), which tries to evict the least recently used cache line, and Random, which evicts a random cache line. In a set-associative cache, the replacement policy is applied separately to each set. For example, consider a 4-way set-associative cache which stores the cache lines \(a, b, c\) and \(d\). Assume that these cache lines have been accessed in the order \(b, a, c, d\), with \(b\) being the least recently used cache line and \(d\) the most recently used, as shown in Fig-

16
When a new cache line \( e \) needs to be stored, \( b \) is evicted to make space for \( e \). Now the cache set contains \( e, d, c \) and \( a \) with \( e \) being the most recently used cache line and \( a \) the least recently used.

\[
\text{least recently used} \quad \downarrow \quad \text{store } e \quad e \quad d \quad c \quad a
\]

Figure 2.3. The LRU replacement policy

2.2 Categories of cache misses

A well-known classification of cache misses, introduced by Hill and Smith [24] categorizes cache misses on single-core systems into “three Cs”:

- **Compulsory misses** (also known as cold misses) occur when a cache line is accessed for the first time, when it cannot be in the cache.
- **Capacity misses** are caused by the cache line being evicted by a previously accessed cache line due to limited cache size. Capacity misses would be cache hits if the cache were infinitely large.
- **Conflict misses** happen in set-associative caches. They are triggered by the cache line being evicted by another cache line mapped to the same set. Conflict misses do not occur in a fully associative cache.

Multi-core systems exhibit all these categories of cache misses. In addition, when cores share data, it affects cache performance in both the shared and private caches. On the one hand, the shared cache reduces the number of accesses to the main memory, since data that is brought into the shared cache by one core can subsequently be accessed by other cores without going to main memory. On the other hand, whenever a core modifies shared data, copies of that data in other private caches are invalidated. When those copies are later accessed by their cores, this triggers a fourth kind of cache miss - known as **coherence misses** [22] (sometimes also known as communication misses) in the private cache of that core, forcing the data to be fetched from the shared cache, or even the main memory.

2.3 Cache performance of programs

Since the purpose of caches is to reduce the time used for a program’s memory accesses, a useful metric for cache performance is the **cache miss ratio**, which is the percentage of memory accesses resulting in cache misses. The lower the cache miss ratio, the better the cache performance. To evaluate the performance of a cache efficiently, we need methods that can estimate the cache miss ratio under different cache configurations at low cost.
Existing methods for estimating the cache miss ratio of a program in caches at different levels include: 1) hardware performance counters 2) cache simulation and 3) software profiling based cache analysis.

In architectures that support hardware performance counters for cache misses, one can directly use these counters to measure the cache miss ratio on actual hardware. These low level counters usually have small measurement overhead and provide accurate results. However, this method can only be used to estimate the cache miss ratio for the current cache configuration. Estimating the cache miss ratio for another cache configuration is difficult since it is often not possible to reconfigure the cache.

Cache simulators [32] [11] [37] mimic the functioning and timing behavior of caches. By simulating the memory accesses, each access is categorized as a cache miss or cache hit. By dividing the number of cache misses by the total number of accesses, we get the cache miss ratio. Cache simulation overcomes the drawbacks of methods based on performance counters since the cache configurations can be easily changed by changing the cache parameters in the simulator. However, simulation usually slows down the execution by hundreds of times. In addition, the simulation must be rerun for each cache configuration, which could take a prohibitively long time.

Software profiling based cache analysis first profiles a program to obtain characteristics related to its cache performance [20] [47] [18] [5]. By feeding these characteristics and the cache configuration to a cache performance model, we estimate the cache miss ratio of the program in the cache under specific configurations. Estimating the cache miss ratio of another cache configuration just requires feeding the model with a different set of inputs corresponding to the new cache configuration and using the model again.

2.4 Software profiling for estimating cache miss ratios

Let us discuss how software profiling can be used to estimate miss ratios for cold misses, capacity and conflict misses, and coherence misses.

Estimating cold misses

Cold misses cannot be avoided unless the cache line is prefetched. Therefore, an over-approximation of the number of cold misses can be obtained as the number of cache lines accessed, i.e., the memory footprint. The effect of prefetching may vary depending on the hardware and software prefetching mechanism in the system.
Data locality and capacity misses

Since a cache has limited size, the cache miss ratio of a program heavily depends on its temporal locality, i.e., recently accessed data is likely to be accessed again. Therefore the cache should exploit the temporal locality of programs and maximize the number of cache hits.

In order to estimate the cache miss ratio, we need some metrics to express the temporal locality of the memory accesses in a program. We start by looking at the *reuse interval* of a memory access, which is the sequence of memory accesses between the previous access to the same cache line, and this access. For example, in Figure 2.4, the sequence of accesses between $t_2$ and $t_8$ including both ends is the *reuse interval* of the memory access to cache line $a$ at $t_9$. There are two commonly used metrics for temporal locality related to the reuse interval.

- The *stack distance* of a memory access is the number of *unique* cache lines accessed during the memory access’s reuse interval. For example, in Figure 2.4, the stack distance of the access to cache line $a$ at $t_9$ is 3.
- The *reuse distance* of a memory access is the number of cache lines accessed during the memory access’s reuse interval. The reuse distance of the access to cache line $a$ at $t_9$ is 7.

In a fully associative LRU cache, the stack distance of a memory access can be used to decide whether it results in a cache hit or miss by comparing its stack distance with the cache size. If the stack distance is smaller than the cache size, the cache would be able to fit all cache lines accessed during the reuse interval without evicting the cache line to reuse. Thus the access would be a cache hit. Otherwise if the stack distance is no smaller than the cache size, the access would result in a cache miss. In Figure 2.4, since the stack distance of the access to $a$ at $t_9$ is 3, if the cache size is larger than 3, the access to $a$ at $t_9$ will be a cache hit.

The stack distance metric was first proposed by Mattson et al in [35] and used by other researchers to analyze program locality [52] [51] [13] [12]. Stack distance based cache analysis [6] takes the *stack distance distribution* of a memory trace as input. The stack distance distribution is a mapping from possible stack distance to the percentage of memory accesses with that stack distance. For a fully associative LRU cache, the cache miss ratio can be calculated by summing up the proportion of memory accesses with a stack distance no smaller than the cache size.

![Figure 2.4. Reuse interval](image)
Stack distance distributions have also been used to estimate the cache miss ratio for caches other than fully associative LRU caches. For instance, Sen and Wood [47] proposed a stack distance-based modeling framework to estimate the cache miss ratio for caches with different sizes, associativities and replacement policies.

To collect the stack distance distribution of a memory trace, one needs to track the memory accesses to unique addresses. The naive method has time complexity of $O(NM)$ and the state of the art algorithm has time complexity $O(N \log M)$ [1], where $N$ is the length of the memory trace and $M$ is the total number of unique cache lines. Sen and Wood [47] showed how the stack distance distribution per set can be collected with special hardware support.

Compared to the stack distance distribution, collecting the reuse distance distribution of a program is less expensive. It takes almost linear time to collect without sampling. Berg and Hagersten [4] developed a sampling based method which has only 40% overhead of the native run. Eklov et al [18] proposed a method called StatStack which estimates the stack distance distribution of a program using the reuse distance distribution and then uses the resulting stack distance distribution to estimate the miss ratio of a fully associative LRU cache.

Estimating cache coherence misses

To take cache coherence misses into account, one needs to consider the additional configuration parameters such as the number of cores that share data. Schuff et al [46] presented a sampling-based approach which records when a thread writes to a shared cache line. This approach estimates the total number of cache misses in the shared and private caches without identifying the coherence misses. Berg et al [5] also presented a sampling-based method that captures coherence misses. To trace coherence misses, a cache line is monitored until it is reused by the same core. During the monitoring, it maintains a writer list of other cores’ writes to the cache line. On reuse by the same core, a non-empty list means that the cache line has been invalidated, triggering a coherence miss. Both of these methods rely on capturing the exact thread interleaving and neither can cope with varying cache sizes and varying numbers of cores.
3. Locks and lock contention

To utilize the cores on a multi-core system, programmers usually write multi-threaded programs. There are several obstacles for such multi-threaded programs to achieve full utilization. First, most programs contain a serial fraction which has to run sequentially, thereby limiting scalability according to Amdahl’s law. Second, the parallel threads may compete for hardware resources such as memory buses and caches, which causes contention, thereby increasing the execution time. Third, when the parallel threads share data, then the accesses to shared data must be synchronized, which incurs extra cost. Mutually exclusive access to shared data is typically ensured by locks. When a thread tries to access a lock currently held by another thread, it has to wait until the lock is available. This waiting time, commonly known as lock contention, is wasted and increases the execution time of the thread.

An important factor determining lock contention is the temporal access pattern for how the threads access a critical section. In an ideal scenario, the accesses do not conflict in time, resulting in no lock contention. Figure 3.1b shows such a scenario where threads access their critical sections one after another. In a worst case scenario, the accesses all occur at the same time, as shown in Figure 3.1a. This serializes the executions of the critical sections, and causes significant contention. The typical case is usually somewhere between the ideal and the worst case.

![Figure 3.1. Lock accesses and lock contention](image)

While the timing of lock accesses decides if there is lock contention, the amount of lock contention depends on both the length of critical sections and the employed lock implementation.
3.1 Lock performance issues

In order to understand the overheads incurred by different lock implementations, let us start by looking at a simple spin lock based on the atomic test-and-set (TAS) operation and its performance issues.

A TAS lock uses the hardware supported atomic instruction test-and-set to implement a lock. Figure 3.2 shows the lock and unlock functions of such a lock. It is based on the atomic test-and-set instruction which sets the value at lock to 1 provided that its current value is 0. The lock is free if its value is 0, and taken if its value is 1. The lock function repeatedly reads the value of the lock until it is free, and then atomically sets its value to 1, thereby taking the lock. To unlock, it simply resets the value at lock to 0.

```
tas_lock (boolean* lock)
{
    while (test-and-set (lock) == 1);
    // spin
}

tas_unlock (boolean* lock)
{
    *lock = 0;
}
```

*Figure 3.2. A TAS-based spin lock*

Let us look at a scenario of three threads accessing a shared TAS lock. In Figure 3.3, thread0, thread1 and thread2 all try to access a TAS lock at time \( t_0 \). Only thread0 succeeds and enters its critical section after experiencing a lock overhead (shown as LO in Figure 3.3), which is the cost of one test-and-set operation. While thread0 executes its critical section between \( t_1 \) and \( t_2 \), thread1 and thread2 keep performing test-and-set operations. When the lock is released by thread0 at \( t_2 \), thread1 successfully performs a TAS operation and enters its critical section. Eventually, thread2 also manages to acquire the lock and finish its critical section.

With the help of this TAS lock access scenario, we summarize some common performance problems of lock implementations.

1. **Problem 1: Memory bus traffic**
   In some lock implementations such as TAS locks\(^1\), threads spend their lock contention repeatedly checking whether the lock is free again. This generates excessive traffic on the memory bus.

2. **Problem 2: Inefficient data access**

\(^1\)On some architectures such as X86, the atomic test-and-set instructions are sometimes replaced by test-and-test-and-set to optimize for performance. Compiler optimizations may also avoid generating memory bus traffic.
Figure 3.3. An example scenario where three threads access a non-delegation lock - LO: lock overhead, CS: critical section

Private caches are faster to access than shared caches and main memory. For performance reasons, it is preferable to access the data from private caches. For instance, TAS locks always read data from main memory, failing to take advantage of the cache.

3. Problem 3: Cache coherence traffic and cache coherence misses
Even if we could solve Problem 2 by accessing shared data from the private cache, whenever we move the shared data between cores, this causes cache coherence traffic and cache coherence misses.

4. Problem 4: Bursty accesses on lock release
In some lock implementations, all waiting threads try to acquire the lock once the lock is released. This typically causes excessive traffic on the memory bus and makes the lock poorly scalable.

3.2 Performance of some lock implementations
In this section, let us consider some popular lock implementations to see how they address the performance problems mentioned in the previous section. Depending on which thread can execute a critical section, we divide them into two categories: delegation locks and non-delegation locks. In a non-delegation lock, a thread always executes its own critical section while in a delegation lock, one thread can also execute critical sections of other threads.

In delegation locks, the lock holding thread can reuse the shared data since it executes multiple critical sections. This makes better use of the private caches, and minimizes the number of cache coherence misses since there is little data movement from one core to another. Although these features bring performance benefits when the lock is highly contended, delegation locks require
threads to pass their critical section code to another thread, which introduces extra overhead. Section 3.2 discusses the performance of delegation locks in more detail.

Non-delegation locks

In this section, we discuss 6 non-delegation locks and how they address and suffer from the previously mentioned performance problems.

**TTAS lock** addresses Problem 1 and Problem 2. A TTAS lock improves on a TAS lock by first repeatedly checking the lock value until it seems free and then trying to perform a test-and-set operation to acquire the lock. Since the value of the lock can be cached, a TTAS lock reduces the memory bus traffic compared to a TAS lock. However, it suffers from Problem 3 and Problem 4. When the lock is released, all cached copies of the lock in other cores are invalidated and all waiting threads will try to perform a test-and-set operation. The generated coherence traffic, coherence misses, and bursts of lock requests could also be a potential scalability bottleneck.

**MCS locks and CLH locks** address Problems 1, 2, and 4. In an MCS lock [36], a linked list is formed and maintained for each lock. The head of the list is the thread currently holding the lock. When a thread fails to acquire the lock, it attaches itself to the tail of the list and spins on a local variable. When the predecessor thread is about to release the lock, it sets the local variable of the successor thread so that the successor thread can stop spinning and try to access the lock. The CLH lock [33] [14] uses a similar mechanism. Both MCS and CLH locks avoid generating traffic on the memory bus since each thread spins on a local variable. They also avoid bursty accesses on lock release, since only the successor thread is eligible to access the lock.

**Pthread mutex locks** addresses Problem 1. When a thread fails to acquire the lock, the operating system puts the thread into a sleep queue. Once a lock becomes free, the operating system issues a system call to wake up all waiting threads so they can retry to acquire the lock. The Pthread mutex lock prevents threads from repeatedly checking if the lock is free by putting them into the sleep queue and waking them up when the lock is free. However, it still has the problem of bursty accesses on lock release since all waiting threads are woken up and all of them will try to acquire the lock at the same time. When there is low contention on the lock, Pthread mutex locks are scalable due to their low overhead. For the high contention case, putting threads to sleep and waking them up brings too much overhead and makes it unscalable.

**Hierarchical backoff locks (HBO) [42]** addresses Problem 1 and Problem 2. When a thread fails to acquire the lock, it sets a backoff time and waits until the time is up before retrying to acquire the lock. This backoff mechanism reduces the memory bus traffic since it prevents threads from re-
peatedly checking the lock. The length of a thread’s backoff time depends on its “distance” to the current lock holding thread, which is architecture specific. By setting a shorter backoff time for nearer threads, these threads are favored when handing over the lock. On a Non-uniform memory access (NUMA) architecture, the nearby threads are those that share a NUMA node. Such a design makes more efficient use of the memory system since nearby threads are likely to share a cache and the shared data can be reused.

Cohort locking [16] addresses Problem 2. There are two levels of locks in cohort locking: a set of threads share a local lock and all threads share a global lock. A cohort lock is locked if and only if the global lock is locked. When a thread acquires a cohort lock, it needs to acquire the local lock first, then the global lock. When releasing a global lock, the current lock holding thread checks if any threads sharing a local lock with the current thread is waiting. If so, it passes the global lock to that thread. Cohort locks exploit the locality of NUMA systems by favoring local threads as the next lock holders.

Delegation locks

Non-delegation locks are widely used due to their simple design, low overhead and high performance when there is little lock contention. In the case of high lock contention, delegation locks typically have better performance. Figure 3.4 shows a scenario where three threads accessing a shared delegation lock at the same time $t_0$. Only thread$_0$ successfully acquires the lock; thread$_1$ and thread$_2$ delegate their critical sections to thread$_0$ with an associated delegation overhead (shown as DO in Figure 3.4). They then wait for thread$_0$ to execute their critical sections and return the results.

![Diagram](image_url)

*Figure 3.4. An example run of a non-delegation lock - LO: lock overhead, DO: delegation overhead, CS: critical section*
In delegation locks, since the lock holding thread can execute other threads’ critical sections, it can reuse the shared data in critical sections. The shared data is likely to be in the thread’s private cache after executing its own critical section. This allows the data of successive critical sections to be fetched from the private cache. A consequence of reusing cached shared data is that there are few cache coherence misses since there is little data movement, causing shorter execution time in the critical section. Delegation locks also avoid generating excessive memory traffic and bursty lock accesses at lock release since the only operations a thread performs is delegating its critical section. To allow the critical section delegation, there is extra overhead involved to communicate the operations in the critical section. When the locks have low contention, this overhead may overshadow the benefits of delegation locks.

Here are two examples of delegation locks.

**Flat combining** [21] addresses Problems 1, 2, 3, 4. Each shared data structure \( D \) used in a critical section is assigned a publication list containing threads’ operations on the data structure. When a thread accesses a lock to operate on \( D \) for the first time, it creates a publication record with its intended operation and inserts it into \( D \)’s publication list. Later when a thread tries to acquire a lock to operate on \( D \), it first updates its own publication record in \( D \)’s publication list, then tries to acquire the lock. If the thread successfully acquires the lock, it becomes the *combiner* which is responsible for combining and executing all operations of all publication records in \( D \)’s publication list. Otherwise the thread waits for a combiner to execute its operation.

**Queue delegation locks (QD locks)** [27] address Problems 1, 2, 3, 4. When multiple threads compete for a lock, the thread that wins the competition becomes the *helper* thread and opens a delegation queue. All other threads trying to access the same lock insert their critical sections into the delegation queue and waits for the helper thread to execute their critical sections. When the helper thread finishes executing a thread’s critical section, it signals the corresponding thread so that it can continue its execution. In the case that a thread’s execution after a critical section does not depend on the critical section results, QD locks allow each thread to continue executing without having to wait for its critical section to finish, which further improves the lock performance.
4. Analytic modeling

This chapter introduces two analytic models - queueing networks which are used in Paper III and Paper IV and Markov chains which are used in Paper I. Other analytic models such as Stochastic Petri Nets can also be used to estimate system performance but they are not used in this thesis.

Compared to other performance analysis methods such as simulation and direct measurement, analytic models represent the target system at a relatively high level of abstraction. They establish a quantitative relationship between relevant system parameters and performance metrics. Analytic models can usually estimate the performance faster than simulation, sometimes at the cost of accuracy. A disadvantage of analytic modeling is its limitation in handling complicated system structures since it sometimes needs to make unrealistic assumptions or approximations to simplify the analysis. However, it does provide insight into the system performance with a small cost, which makes it a competitive alternative to direct measurement and simulation. The challenge is to build an analytic model that not only describes the characteristics of the system, but also can be analyzed within a reasonable amount of time.

In Section 4.1, we introduce the basic concepts of discrete-time Markov chains. In Section 4.2, we discuss queueing networks and how to analyze them.

4.1 Discrete-time Markov chains (DTMC)

A random variable is a variable whose value is affected by randomness. For example, a stock market index or the air temperature can be seen as random variables. A discrete-time stochastic process is a sequence of random variables. For example, the stock market index at 10:00 every weekday since 1990, or the air temperature in Uppsala every 30 minutes since 8:00 this morning. A discrete-time Markov process is a discrete-time stochastic process with the memoryless property. This property states that the value of the next variable in the sequence only depends on the value of the current variable. In other words, a discrete-time stochastic process \( X_0, X_1, X_2, \ldots \) is a Markov process if for all \( n \), the probability distribution of \( X_{n+1} \) only depends on the value of \( X_n \), regardless of the values of \( X_0, \ldots, X_{n-1} \).

The state space \( \Sigma \) of a Markov process is the set of values the variable can take, which can be either finite or infinite. If the state space is finite or countable, the resulting Markov process is a Markov chain.
A discrete-time Markov chain with a finite state space, can be represented by a directed graph, whose nodes are the states and an edge between state $s$ and state $s'$ represents the conditional probability of the next variable in the sequence being $s'$ given that the current variable is $s$. The graph can be represented as an $n \times n$ matrix, known as the transition matrix. This matrix sometimes can depend on the variable index $i$. Each element $p_{s,s'}^{(i)}$ in a transition matrix $p^{(i)}$ denotes the conditional probability $P(X_{i+1} = s'|X_i = s)$.

The distribution of the first random variable $X_0$ is given by an initial probability distribution, denoted as $P^{(0)}$, i.e., the probability $P^{(0)}(s)$ defines the probability of $X_0$ being in state $s$. Given the initial probability distribution and transition matrix, one can calculate the probability distribution for $X_i$. Due to the memoryless property, for each state $s$, the probability of $X_i$ being in any state $s$ can be calculated as the weighted sum of all the probabilities of transitioning from another state $s'$ to $s$ at $X_{i-1}$, for all $i > 1$

$$P^{(i)}(s) = \sum_{s' \in \Sigma} P^{(i-1)}(s') \cdot p_{s',s}^{(i-1)}.$$  \hspace{1cm} (4.1)

By using equation 4.1 repeatedly, we can calculate the probability distribution for any $X_i$ given the initial probability distribution and transition matrices:

$$P^{(i)} = P^{(0)} \cdot \prod_{0 \leq j \leq i-1} p^{(j)}.$$  \hspace{1cm} (4.2)

If the transition matrix $p^{(i)}$ is independent of $i$, the Markov chain is called a time homogeneous Markov chain. For many time homogeneous Markov chains, the probability distribution converges to a certain distribution, which is independent of the initial probability distribution. This converged distribution $\pi$, known as the stationary (or steady-state) distribution, can be calculated as the solution to the equation $\pi = \pi \cdot p$.

Markov chains are often used to study the evolution of systems. Let us take the gambler’s ruin problem as an example.

![Markov chain modeling the gambler’s ruin problem](Figure 4.1)

**Example: gambler’s ruin program**

A gambler starts with $m$ dollar(s) and performs a sequence of bets. Each time he bets, he gains one dollar with probability $p$ and loses one dollar with prob-
ability $1 - p$. The game stops when the gambler either wins (reaches a goal of $n$ dollars) or gets ruined (0 dollars left).

This problem can be modeled as a discrete-time Markov chain shown in Figure 4.1. For all $1 \leq i \leq n - 1$, the state $i$, which represents the gambler having $i$ dollars, has an edge to state $i + 1$, representing the probability of gaining one dollar. It also has an edge to state $i - 1$, representing the probability of losing one dollar. Once the Markov chain reaches state 0 or state $n$, it will stay in that state forever since the game is over.

Initially, the Markov chain starts at state $m$, representing the gambler having $m$ dollars. The probability distribution after any number of bets can be calculated using Equation 4.2. In particular, the probability $P^{(i)}(n)$ represents the probability of winning in at most $i$ steps. With the probability distributions, we can answer questions about the bets. For example, if we start with 3 dollars ($m = 3$), the probability of winning at each bet is 50% ($p = 0.5$) and the goal is to win 10 dollars ($n = 10$), the probability of getting ruined within 10 bets is 0.34.

The winning probability is then $\lim_{i \to \infty} P^{(i)}(n)$, which we abbreviate as $P_{\text{win}}^{m}(n)$.

By simple recursion we have $P_{\text{win}}^{m}(n) = \frac{1 - (1 - p)^m}{1 - (1 - p)^n}$ if $p \neq 0.5$ and $P_{\text{win}}^{m}(n) = \frac{m}{n}$ if $p = 0.5$. In Section 5.1, we use similar ideas using Markov chains to estimate the cache miss ratio.

### 4.2 Queueing networks

Queueing theory is the mathematical study of queues and systems of queues, with the purpose of predicting performance parameters such as queue lengths and waiting times [9]. It is often used to model congested systems with limited resources and analyze their performance. Examples of such systems are telecommunication, traffic, and systems performing customer service. Intuitively, since computer systems often have limited shared resources, queueing theory should be a candidate to evaluate the waiting times at these resources and analyze whether they cause performance bottlenecks.

The basic component of a queueing network is a node, sometimes called a service station. A node has a queue where jobs wait before they get service. A node has three basic parameters: service time, number of servers, and queueing discipline. The service time is the time it takes to serve one job. The service time has a specified probability distribution, which can be arbitrary, but the most commonly used one is an exponential distribution. The number of servers specifies the number of jobs the node can serve simultaneously. For example, a one-server node can only serve one job at a time. An infinite-server node can serve infinitely many jobs simultaneously. If there are more jobs than servers, only a limited number of jobs will be served while other jobs must wait for their turn. The queueing discipline is the policy deciding
in which order jobs are served. Commonly used queueing disciplines include First Come First Served (FCFS), Round Robin (RR) and Service In Random Order (SIRO).

A set of nodes can be connected into a queueing network. Each node can have its own service time, number of servers and queueing discipline. After visiting one node, jobs can arrive at another node to get service. The likelihood of visiting one node after another is modeled probabilistically. In a queueing network with $N$ nodes, an $N \times N$ matrix $r$, called a routing matrix is used to specify these probabilities. Each element $r_{ij}$ in the routing matrix $r$ represents the probability of arriving at node $j$ after departing from node $i$.

Types of queueing networks
A queueing network can be either open or closed. The main difference between these two types is that in an open queueing network, jobs can arrive from the environment and leave the queueing network, whereas in a closed queueing network, jobs always stay in the network so that the number of jobs stays constant. Figure 4.2 shows an example of a closed queueing network, which is called the machine repairman model. It models a factory where $K$ machines are put to use during some time until breaking down, get repaired and then put to work again. There are two nodes in the queueing network: a work node and a repair node. Jobs, which represent machines, visit the work and repair nodes repeatedly. The work node is an infinite-server node since the machines work in parallel. The service time at the work node models how long a machine can work until it breaks down. The repair node has only one server, indicating that machines can only be repaired one at a time. Its service time represents the time it takes to repair a machine. We assume a FCFS queueing discipline at the repair node.

In open queueing networks, jobs can arrive at the network from outside and leave the network after being served. Since jobs arrive from outside the network, we need to specify the arrival pattern of jobs. Typically, it is assumed that jobs arrive as a Poisson process with a specified arrival rate, commonly denoted as $\lambda$. Figure 4.3 shows an example open queueing network of one node with $m$ servers. A standardly used notation for describing the parameters of such a network was introduced by Kendall [26], known as Kendall’s notation:
$A/S/c/K/N/D$ where $A$ indicates the arrival distribution of jobs, $S$ the distribution of the node’s service time, $c$ the number of servers in the node, $K$ the maximum number of customers allowed in the network, $N$ the total number of jobs and $D$ the queueing discipline. For example, a $M/M/1/\infty/\infty/FCFS$ queueing network specifies a one-server node with exponential inter-arrival time, exponential service time, an infinite population of jobs with a FCFS queueing discipline.

**Multi-class queueing networks**

Sometimes different jobs follow different patterns of visiting the nodes. They may also need different amount of time for getting service at the same node. In the terminology of queueing theory, they have different routing matrices and/or service times. A generalization of queueing networks - multi-class queueing networks, can describe such differences.

![Figure 4.3. An example service station](image)

**Performance metrics**

If we let the queueing network keep running, as time goes to infinity, performance metrics such as the average queue length and average waiting time at the nodes may converge, similar to the convergence of the probability distribution of Markov chains. If such a convergence happens, the queueing network is said to reach a “steady state”.

We are often interested in the following performance metrics when the queueing network is at steady state:

- average queue length ($Q_i$), which is the average number of jobs at node $i$,
- average waiting time ($W_i$), which is the time a job waits at node $i$’s queue.
- average response time ($R_i$), which is the average time a job spends from reaching the queue of node $i$ to leaving the node. This is the sum of the average waiting time and the service time.

These metrics can be used to analyze how congested the system is. For example, if the waiting time is dominant in the response time, a job spends much more waiting to get served than the service time, meaning the system is congested. An important relationship between the average queue length, arrival rate and average waiting time is formulated by Little’s law [29]:

$$Q = \lambda W$$  \hspace{1cm} (4.3)

This law states the average queue length can be calculated as the product of arrival rate and average waiting time.
Calculating performance metrics for closed queueing networks

In this thesis, we only consider closed queueing networks since we model multi-threaded programs where the number of threads is constant. For queueing networks with exponential inter-arrival time and service time with FCFS queueing disciplines, the performance metrics can be calculated using several methods. The convolution algorithm and mean value analysis are two of the most popular ones [9]. In this thesis, we use the mean value analysis (MVA) to analyze our queueing networks.

Mean value analysis (MVA) [43] is a recursive method of calculating the performance metrics of a closed queueing network. It is based on the observation that in a queueing network with \( k \) jobs, at steady state, a job arriving at a node observes the rest of the network as if it had \( k - 1 \) jobs in steady state. This observation is known as the Arrival Theorem. Using this observation, the performance metrics of the system with \( k \) jobs can be derived from those with \( k - 1 \) jobs. For a queueing network with one job, the average waiting time at any node is 0 since there are no other jobs competing for service.

In a closed multi-class queueing network with \( R \) classes, the number of jobs in each class is represented by a population vector \( K = (K_1, K_2, \ldots, K_R) \) where \( K_r \) is the number of jobs in class \( r \). The performance metrics in this queueing network can be derived from the performance metrics for networks with one fewer job in each class. To calculate the average queue length, waiting time at each node with population vector \( K \), we need to calculate these performance metrics for \( K_1 = (K_1 - 1, K_2, \ldots, K_R) \), \( K_2 = (K_1, K_2 - 1, \ldots, K_R) \), \ldots, \( K_R = (K_1, K_2, \ldots, K_R - 1) \). Thus the time complexity of MVA is \( \mathcal{O}(N \cdot \prod_{1 \leq r \leq R} K_r) \), since we need to calculate performance metrics for the smaller population vectors.
In this chapter, we present techniques for using the reuse distance distribution of a program to predict its cache miss ratio. While the reuse distance distribution is an easy-to-collect metric which describes temporal data locality, it provides rather weak information about the temporal locality. This makes it challenging to use the reuse distance distribution as a basis for predicting the cache performance under different cache configurations.

In Section 5.1, we present a general modeling framework for estimating the cache miss ratio under different cache configurations based on the reuse distance distribution. In all caches, we need to estimate the cache miss ratio including cold misses, capacity misses and conflict misses.

In Section 5.2, we present our probabilistic models for estimating the number of cache coherence misses. In private caches of modern multi-core systems, we would like to estimate the coherence misses since it brings insight into the cost of data sharing.

5.1 A cache modeling framework

One of the goals of our technique is to use only reuse distance distributions for estimating cache miss ratios under a variety of cache configurations of different sizes, associativities and replacement policies. So far, reuse distance distributions have only been used for predicting miss ratios for fully associative LRU and RANDOM caches [5] [18]. Furthermore, it is difficult to generalize the models proposed in [5] and [18] to cache replacement policies found in modern architectures such as PLRU and bit-PLRU. Another goal of our technique is to have a general framework for cache performance prediction, which can be instantiated for different cache configurations, such as various replacement policies and associativities.

Let us first specify the problem to be solved. Assume we have a program with fixed input data, which induces a fixed sequence of memory accesses. If we only know the reuse distance distribution of the sequence of memory accesses, what would be the cache miss ratio in caches with different configurations including different cache sizes, associativities and replacement policies?

To obtain a solution, we consider a random memory access and try to estimate the probability of it being a cache miss. Figure 5.1 shows a memory...
access sequence where the cache line $a$ is first accessed at $t_0$ and again accessed at $t_5$. We want to estimate the probability of the access to $a$ at $t_5$ being a cache miss, based on the information provided by the reuse distance distribution. A natural approach is to study the reuse interval from $t_1$ to $t_4$, and use the information provided by reuse distance distribution to estimate the probability that the access at $t_5$ is a miss. Whether or not $a$ is evicted before $t_5$ depends on the exact sequence of accesses during the reuse interval and the cache state at $t_5$. It is clearly not possible to calculate the probability of each possible sequence of memory accesses during the reuse interval in order to obtain the probability of a miss, using only information provided by the reuse distance distribution. Instead, we must extract some key properties of the reuse interval in order to develop a method which is both tractable and reasonably accurate.

We propose a method which at each time point during the reuse interval summarizes information about the cache contents that is relevant for whether the reuse of $a$ will be a hit or miss. The summarized information is represented as an (abstract) state of the cache. Then we construct a Markov chain describing the evolution of this state during the reuse interval. By studying the evolution of the state, we estimate the miss probability.

\[ \text{reuse interval of access to } a \text{ at } t_5 \]

\begin{figure}[h]
\centering
\begin{center}
\begin{tikzpicture}
    \node[circle,draw] (c) at (0,0) {$c$};
    \node[circle,draw] (b) at (1,0) {$b$};
    \node[circle,draw] (a) at (2,0) {$a$};
    \node[circle,draw] (t0) at (2,-0.5) {$t_0$};
    \node[circle,draw] (t1) at (3,-0.5) {$t_1$};
    \node[circle,draw] (t2) at (4,-0.5) {$t_2$};
    \node[circle,draw] (t3) at (5,-0.5) {$t_3$};
    \node[circle,draw] (t4) at (6,-0.5) {$t_4$};
    \node[circle,draw] (t5) at (7,-0.5) {$t_5$};
    \node[circle,draw] (a) at (8,0) {$a$};
    \node[circle,draw] (d) at (9,0) {$d$};
    \node[circle,draw] (e) at (10,0) {$e$};
    \draw (c) -- (b) -- (a) -- (t0) -- (t1) -- (t2) -- (t3) -- (t4) -- (t5) -- (a) -- (d) -- (e);
    \draw (t0) -- (t1) -- (t2) -- (t3) -- (t4) -- (t5);
    \draw (t1) -- (t2) -- (t3) -- (t4) -- (t5);
    \draw (t2) -- (t3) -- (t4) -- (t5);
    \draw (t3) -- (t4) -- (t5);
    \draw (t4) -- (t5);
\end{tikzpicture}
\end{center}
\caption{Reuse interval}
\end{figure}

In our framework, this Markov chain will always contain the states $hit$ and $miss$, with the property that one of these states is reached when the reuse occurs. In addition, the Markov chain has a number of states that are relevant for determining whether the reuse will be a hit or a miss. It is clearly useful with a state $evicted$ to indicate that $a$ has been evicted. Upon finishing the construction of the Markov chain, we assign an initial probability distribution over the states right after $t_0$. By standard analysis of the Markov chain, we can estimate how this probability distribution evolves with each memory access during the reuse interval. In detail, the Markov chain contains the following components:

- a set $\Sigma$ of states, which must contain the states $hit$ and $miss$.
- an initial probability distribution, denoted $P^{(0)}$ over the state space $\Sigma$, which for each state $s \in \Sigma$ defines the probability $P^{(0)}(s)$ of being in state $s$ right after $t_0$.
- transition probabilities, denoted $p^{(i)}_{s,s'}$, which for each $i = 0, 1, \ldots$ and each pair of states $s, s' \in \Sigma$, defines the conditional probability of being in $s'$ at $t_{i+1}$, given that the Markov chain is in $s$ at $t_i$. In some cases, the transition probabilities may depend on $i$. 

34
We can calculate the probability distribution at any time point $t_i$ with the techniques introduced in Section 4.1. Given a Markov chain that models the evolution of the cache as above, we can now for $i = 0, 1, 2, \ldots$ calculate the probability distribution, denoted $P^{(i)}$, over its states at time point $t_i$. For $i = 0$, $P^{(0)}$ is the initial probability distribution $P^{(0)}$. For $i = 1, 2, \ldots$, we calculate the probability distribution using the formula

$$P^{(i)} = P^{(0)} \cdot \prod_{0 \leq j \leq i-1} p^{(j)}$$

(see Equation 4.2).

We can estimate the cache miss probability of any reuse distance $i$ with the probability distribution over the states. The miss probability of a random memory access with reuse distance $i$ is the probability of being in the state evicted when being reused. Using the estimated miss ratios for all reuse distances $i$ and the reuse distance distribution, we can predict the miss ratio of the whole memory access by computing the weighted sum of the miss ratios over all reuse distances. An alternative method is to see that the Markov chain will move towards state hit and state miss as $i$ goes to infinity. We can calculate the miss ratio by calculating the probability of being in the state miss as $i$ goes to infinity. The miss ratio is calculated as $\frac{P^{(i)}(\text{miss})}{P^{(i)}(\text{miss}) + P^{(i)}(\text{hit})}$ for a sufficiently large $i$, as shown in Paper I.

Example: model for the LRU replacement policy

Let us illustrate our framework by applying it to predict the cache miss ratio under the least recently used (LRU) replacement policy and associativity $A$. In order to construct a Markov chain for the LRU policy cache, we must define a set of states which are relevant. A natural start is to define, for each cache line and each position in the access sequence, the age of a cache line as the number of unique number of cache lines accessed between the previous access to the cache line and the current position. The age can be considered as a generalization of stack distance. While stack distance is only defined for the cache line accessed at each position, age is defined for all the cache lines at each position. Thus, right after a cache line $a$ is accessed, its age is set to 0. The age of $a$ increases by one when an older cache line is accessed. For example, in Figure 5.2, cache line $a$, $c$ and $d$’s ages increase by one when accessing a cache line $b$, which was older than all the other three cache lines before its access. When $a$’s age reaches cache associativity $A$, $a$ is evicted since each cache set can only fit $A$ cache lines.

The purpose of the Markov chain for the LRU cache is to follow the age of a specified cache line $a$ during its reuse interval. Initially, right after $a$ is accessed, its age is 0. Thereafter the question is how to assign transition probabilities. Since the age of cache line $a$ increases when accessing an older cache
line, the problem is to estimate the probability that this happens, given only
the reuse distance distribution. In the following, we show that this probabil-
ity is closely related to the reuse distance distribution. Consider an arbitrary
memory access during cache line a’s reuse interval, say cache line b at ti, as
shown in Figure 5.3, we claim that a’s age increases after this access if and
only if the reuse distance of the access is bigger than i. To see why, note that if
the reuse distance of b is bigger than i, then the last access to b was before t0,
implying that b is older than a at ti, which will increase a’s age. Conversely,
if a’s age increases at ti, the access at ti must be older than a, implying its last
access was before ti, therefore having a reuse distance bigger than i.

With this property relating a cache line’s age with reuse distance, we can
construct a Markov chain representing the increasing of age of cache lines,
which may eventually lead to cache line eviction.

Before discussing the resulting Markov chain, we first introduce some no-
tations used in the Markov chain. We use the notation rdd(i) to represent the
probability of the reuse distance distribution being i, rdd(\geq k) for \sum_{i=k}^{\infty} rdd(i),
i.e., the fraction of accesses whose reuse distance is at least k, and write
rdd(> k) for rdd(\geq (k+1)). It will be convenient to also define the marginal
reuse distance distribution (mrdd for short) of a trace as the function mrdd
from natural numbers to probabilities, defined by mrdd(k) = \frac{rdd(k)}{rdd(\geq k)}. That
is, mrdd(k) is the probability that the reuse distance is k, provided that it is at
least k.

Figure 5.4 shows the resulting Markov chain for a LRU cache with associa-
tivity A. There are two types of states: age states (in blue circles) and status
states (in red rectangles). All age states represent the case where the cache line
a is still in the cache. The status states are hit (a reused and results in a hit),
miss (a reused and results in a miss) and evicted (a has been evicted but not yet
reused). Each age state has an edge to the hit state, indicating that a is reused
with a probability mrdd(i). The age is monotonically non-decreasing, there-
fore there is only an edge to a bigger age or the current age. To increase the
age of \( a \) at \( t_i \), an access with reuse distance bigger than \( i \) is needed according to the previous property, with probability \( \text{rdd}(\geq i) \). Since the probability of all outgoing edges in a Markov chain sums up to 1, the probability of staying with the same age is \( 1 - \text{rdd}(>i) - \text{mrdd}(i) \).

Note that the transition probability \( p_{s,s'}^{(i)} \) for the LRU cache not only depends on the states \( s \) and \( s' \), but also on the index \( i \) of the current step \( t_i \). This makes our Markov chain not time-homogeneous.

When the cache line has been evicted, i.e., in state `evicted`, another reuse of \( a \) will cause a cache miss. Both states `miss` and state `hit` are absorbing states.

Figure 5.4. Markov chain for a LRU cache with size \( A \)

We can now calculate the cache miss ratio using our Markov chain. The initial probability is 1 for state 0 and 0 for all other states. With the initial probability distribution and transition probabilities defined in the Markov chain, we can calculate the probability distribution at any reuse distance \( i \) then its cache miss ratio using the method in the general framework.

In Paper I, we show how analogous Markov chains can be constructed for other policies, including Random, PLRU and bit-PLRU. Some transitions in these Markov chains are triggered by an access being a cache miss. For this case, we introduce an unknown variable \( x \) to represent the sought miss ratio, and let some transition probabilities \( p_{s,s'}^{(i)} \) depend on \( x \), so that \( P^{(\infty)}(\text{miss}) \) in general depends on \( x \). The sought miss ratio will then be defined by an implicit equation of the form \( x = \frac{P^{(\infty)}(\text{miss})}{P^{(\infty)}(\text{miss}) + P^{(\infty)}(\text{hit})} \), which we can solve by standard methods (e.g., fixpoint iteration).

In set-associative caches, when estimating the probability of an access to a random cache line \( a \) being a cache miss, we need to consider only those accesses during \( a \)’s reuse interval that are mapped to the same set as \( a \). We define the set reuse distance of a memory access as the number of cache lines accessed during the memory access’s reuse interval that are mapped to the same set as the memory access. Since the only information we have is the reuse distance distribution, we must estimate the set reuse distance distribution from it. For the analogous problem of estimating set stack distance distributions from stack distance distributions, Hill and Smith proposed a probabilistic
model based on the assumption that each cache line is mapped randomly to a set, and that the mappings of two different cache lines are independent [24]. In our benchmarks, we found that the probability of two cache lines mapping to the same set also depends on the set mapping function. Paper I propose a model which improves in accuracy over that of [24] by taking characteristics of the mapping function into account.

Evaluation
We evaluated our models using the SPEC 2006 benchmark suite [23] under 20 cache configurations with varying size, associativity and replacement policy. The cache miss ratios predicted by our models from (set) reuse distance distributions were compared with cache miss ratios obtained from cache simulations with models of the actual cache configurations. The average absolute error of our model over all benchmarks and all cache configurations is 0.72% for LRU caches, 0.93% for PLRU caches and 0.98% for bit-PLRU caches. Sampling the reuse distance distribution added 1% error. Our method for estimating set reuse distance distribution from reuse distance distribution introduces another 2% error. Compared to StatStack, the absolute error of our model is 0.2% lower. The difference is larger for set-associative caches with a small set size.

Related work
Simulation has been used to analyze cache performance. There have been many available simulation tools supporting cache simulation such as Simics [32], CacheSim [11], CacheGrind [37]. However, simulation usually takes orders of magnitudes longer than native execution, which is sometimes prohibitively slow. While sampling [3] [49] [50] could speed up the simulation process, it usually incurs a cost in accuracy.

Nowadays, upon realizing the importance of understanding the cache performance, major chip manufacturers such as Intel and AMD usually support hardware performance counters to help programmers evaluate their programs’ cache performance. Many software tools and libraries such as oprofile [48], PAPI [30] are widely used for performance analysis. While these tools are very accurate in measuring the cache performance, they can only measure the performance for the current configuration.

Previously, several profiled-based analytic models have been proposed to estimate cache performance. Guo and Solihin [20] predicted cache miss ratios for varying cache replacement policies, cache sizes and associativities, based on a combination of stack distance and reuse distance, which is more expensive to collect than only the stack distance distribution. Sen and Wood [47] developed an online modeling framework to predict the cache miss ratio un-
der different configurations based on set stack distance distribution. They also showed how set stack distance distribution can be obtained by special on-chip hardware, which however is not available in today’s processors.

5.2 Estimating cache coherence misses

Multithreading is necessary for achieving performance on multi-core systems. Typically, threads share data. When the work load among threads is balanced, each thread is usually pinned to a designated core to maximize core utilization. In this case, the data shared by threads is also shared by cores. In modern architectures where cores have private caches, data sharing may cause cache coherence misses in the cores’ private caches due to the cache coherence protocol. When one core writes to shared data, this invalidates all copies of the shared data in other cores’ private caches. When that data is later accessed on other cores, it may lead to a coherence miss since the data has been invalidated in its private cache.

The number of coherence misses indicates whether data sharing causes performance problems in the private caches. Being able to estimate the number of coherence misses allows programmers to decide on how to distribute code and data over the cores to optimize for private cache performance.

Estimating the number of cache coherence misses for a running program turns out to be difficult. For instance, even though there are hardware performance counters for some other performance-related metrics such as the total number of cache misses in different cache levels and the number of instructions executed, there is no such counter for coherence misses. One reason is that such a counter would be costly. It would require keeping track of three operations: 1) a cache line being invalidated 2) the same cache line being reused and 3) whether the reuse would have been a cache hit without the invalidation. Keeping such information is both time and space consuming.

In this section, we discuss the ideas of three models for predicting the number of cache coherence misses of a parallel program on multi-core. Two of the models (uniform and phased) are based on characterizing the conditions for triggering coherence misses and estimating the probability that these conditions occur. The input to these models are the program’s the reuse distance distribution and write frequency to shared data, which can be obtained using software profiling. The third model (symmetric) provides a simpler analysis provided the program accesses both the shared and local data in a uniformly random manner. This model does not require any profiling but it relies on the cache performance counters. The detailed models and their evaluations are discussed in Paper II. The rest of this section is organized as follows. Section 5.2.1 studies the conditions for triggering a cache coherence miss. Section 5.2.2 presents our models for predicting the number of cache coherence misses.
5.2.1 Characterizing cache coherence misses

Let us first study the conditions for triggering a cache coherence miss by looking at a scenario shown in Figure 5.5. Assume we have a program running on $N$ cores and all cores share a cache line $X$. At a time point $t$, $Core_i$ accesses $X$, which installs $X$ in $Core_i$’s private cache. At a later point in time $t'$ ($t' > t$), $Core_j$ writes to $X$. This write causes an invalidation of the cache line containing $X$ in $Core_i$’s private cache. When $Core_i$ later accesses $X$ again at $t''$ ($t'' > t'$), the invalidated cache line causes this reuse of $X$ to become a coherence miss.

To summarize, in order for a memory access $x$ to cache line $X$ by $Core_i$ to be a cache coherence miss, the following four conditions must be satisfied:

1. $x$ is not a cold miss, i.e., $x$ is a reuse.
2. $x$ is not a capacity or conflict miss.
3. $X$ must be a shared cache line since only accesses to shared data can trigger the cache coherence protocol to invalidate the cache line in private caches.
4. Another core other than $Core_i$ writes to $X$ during the reuse interval of $x$.

Thus we can estimate the probability of each of these conditions and predict the number of cache coherence misses by combining these probabilities.

5.2.2 Predicting the number of coherence misses

The exact number of coherence misses may vary with the exact interleaving of the threads on cores, which is hard to predict. We therefore propose probabilistic models to predict the number of coherence misses.

The *uniform* model

For programs where the accesses to shared data by different threads occur uniformly and temporally uncorrelated, we propose the *uniform* model. This
model is based on estimating the probability of the four conditions that trigger a coherence miss. We address each of the four conditions as follows.

- Condition 1 and 2: if we combine Condition 1 and 2, we get the condition of the memory access being a cache capacity/conflict hit. This probability can be calculated using existing methods such as methods found in Paper I and [18] based on the reuse distance distribution of the program.

- Condition 3: the condition states the memory access must be to shared cache lines. We can obtain the number of accessed to shared cache lines by profiling each thread of the target program and finding the shared cache lines used by multiple threads.

- Condition 4: we propose to estimate the probability of Condition 4 using the write frequency to the shared data as follows. For a reuse distance $d$ by $Core_i$ to the shared cache line $X$, the probability of another core $Core_j$ not writing to $X$ during the reuse interval of length $d$ is $(1 - f_j)^{d+1}$ assuming the write frequency to shared data is $f_j$. Since the memory accesses of different cores are assumed to be independent, the probability of at least one of the cores other than $Core_i$ writing to $X$ within $d$ is $1 - \prod_{j \neq i} (1 - f_j)^{d+1}$.

When all of the four conditions are fulfilled, a coherence miss is triggered. By combining the probabilities of all four conditions, we can estimate the probability of a memory access being a cache coherence miss. The details of the calculation are discussed in Paper II.

The phased model
The uniform model makes the assumption that all threads/cores access each shared cache lines uniformly throughout the whole program execution. This is often too simplified for a real program. It is common that the shared data is accessed with a different pattern in different phases of the program, usually divided by synchronization. For these cases, we generalize the uniform model by dividing the whole execution into phases, usually separated by synchronization primitives such as barriers, conditional variables, etc. We then account for the coherence misses within each phase with the uniform model and inter-phase coherence misses using a similar method.

The symmetric model
In the symmetric model, we propose a simple analysis of coherence misses, which does not rely on software profiling to obtain reuse distance distribution of shared data and write frequencies. It relies on the rather strong assumption that the threads access both local and shared data in a uniform and symmetric way. More precisely, it assumes that the program processes local data, which is evenly divided between threads. While the threads process local data, they also access shared data. We also assume that the amount of local data is sig-
significantly larger than the amount of shared data, and the pattern of accesses to local and shared data can be considered as uniform and symmetric; in particular, it should stay the same regardless of the number of threads. For shared data, this means that the interleaving of accesses by threads can be taken to be independent and uniformly distributed over threads. An example program that fulfills these assumptions could be a network packet processing multi-threaded program where each thread decompresses the packets and maintains a shared data structure to keep track of them. Since the access pattern is independent of the number of threads, the cache miss ratio of cold capacity and conflict misses are approximately the same when the program runs on \( N \) cores as when it runs on one core. Therefore, the number of cold, capacity and conflict misses of each core when running on \( N \) cores is approximately \( \frac{1}{N} \) of those when running on one core. Since the shared data is also accessed by each core in the same pattern, when a thread accesses some shared data, the probability of the last write to the shared data not being by the same thread is \( 1 - \frac{1}{N} \). Thus, the probability that the shared data is invalidated is simply \( 1 - \frac{1}{N} \). This means the number of cache coherence misses on \( N \) cores is \( (1 - \frac{1}{N}) \cdot M_{1}^{\text{shared hit}}(1) \) assuming that the number of cache hits in shared data on a single core is \( M_{1}^{\text{shared hit}}(1) \).

Let us denote the total number of cache misses for a random core \( \text{Core}_i \) as \( M_{i}^{\text{miss}}(N) \) and the number of cache hits with one thread as \( M_{1}^{\text{hit}}(1) \), we have the following equation

\[
M_{i}^{\text{miss}}(N) = \frac{1}{N} \cdot M_{1}^{\text{miss}}(1) + (1 - \frac{1}{N}) \cdot M_{1}^{\text{shared hit}}(1) \tag{5.1}
\]

It is difficult to estimate \( M_{1}^{\text{shared hit}}(1) \) without furthering profiling the program. One possibility, which we use, is to apply a regression method using the number of cache misses with one core and \( i \) cores, which can be measured using hardware performance counters. With these measured cache misses and equation 5.1, we can estimate \( M_{1}^{\text{shared hit}}(1) \). Then we are able to predict the number of cache coherence misses with \( M_{1}^{\text{shared hit}}(1) \).

Evaluation

We evaluated the uniform, phased and symmetric models with running benchmarks in the PARSEC benchmark suite [7] on 2 – 8 cores. Since it is not possible to directly measure the number of cache coherence misses, we added the number of cold misses and capacity misses, which is estimated with StatStack, with the number of predicted coherence misses and evaluate the accuracy of our models by comparing the estimated total number of cache misses with the measured total number of cache misses. The average relative error for uniform is 5.8% and for phased is 8.02% (phased and uniform were used for different benchmarks). The symmetric model is only applicable for benchmark dedup and its average prediction error of L2 misses over different numbers of cores is
5.4%. Different benchmarks show very different cache behaviors. For details, see Paper II.

Related work
So far the work taking coherence misses into account has been sampling-based. Section 2.4 discusses the work by Schuff et al [46] and Berg et al [5] in detail. Both of these methods rely on capturing the exact thread inter-leavings. Sampling-based approaches are sensitive to interference from other processes. In addition, it cannot be used to model cache misses for another hardware configuration (e.g., different cache size). Our profiling approach only collects software-specific data, which makes our profiling process insensitive to interference. Another advantage of analytical-based approaches including ours is the ability to evaluate performance in another system configuration. For example, our model can be used to predict the number of cache misses with a different cache size.
6. Predicting lock contention

In this chapter, we discuss using queueing models to predict the cost of lock contention in parallel programs on multi-core. An advantage of using analytic models in this case is that they provide insights into the factors contributing to lock contention. They can also quantify such contributions by answering "what if " questions. For example, how much would the lock contention be if the size of all critical sections is reduced by 10%? How much lock contention would there be if the locks were accessed in a different order. We focus on answering the following two questions:

1. Given a parallel program, can we estimate the lock contention on a target multi-core system based on information obtainable from profiling a run on a single core?
2. If the current lock implementation has high lock contention, one way to reduce the contention is to use another lock implementation. Can we forecast if another lock implementation would reduce the lock contention without having to reimplement the program?

We address question 1 by building a model of the lock access patterns of the target program from the profiling run. Such a model can also be used to answer question 2, but we must then develop a different model for the mechanism and overhead of another lock. The rest of the chapter is organized as follows. Section 6.1 presents our queueing network model to predict the lock contention based on the lock access pattern and lock holding times, which can be obtained from profiling a single-threaded run. Section 6.2 summarizes the model for forecasting the lock contention of another lock implementation.

6.1 Predicting the lock contention of non-delegation locks

Lock contention may become a scalability bottleneck for a parallel program that executes on multi-core systems. Quantifying the lock contention and understanding its causes is a first step to optimizing the program and reducing lock contention. One way to obtain the lock contention is to directly measure it on the target multi-core system. However, it is not trivial to develop a tool to measure lock contention accurately without interfering with program execution [10] [25]. Some analytic models such as Stochastic Petri Nets are expressive in describing program synchronizations. However, analyzing a stochastic petri net model is too expensive even for programs with a small number of
locks, as discussed in Paper III. In this section, we investigate the accuracy of using queueing network models to predict the lock contention.

6.1.1 Modeling a program with locks

Before introducing the models in detail, we first discuss one way to view the structure of a parallel programs with locks, which is later used in the models. The code of a parallel program with lock accesses can be divided into two types of code segments (we use $cseg_i$ to denote a code segment $i$): local computation segments and lock segments. We assume that local computation segments contain no synchronization, and that their execution time is independent of the number of used cores. Each lock segment includes accessing a critical section protected by a lock.

For example, Figure 6.1 shows a method which processes and categorizes two types of packets. For each packet $packet_i$, it first processes it, then adds it to either list$_1$ or list$_2$ depending on its type (we assume there are only two types of packets). Each list is protected by a lock. The method $process(packets)$ contains three code segments: two lock segments ($cseg_2$ and $cseg_3$) and one parallel local computation segment ($cseg_1$).

```
method process(packets)

loop
    for each packet:
        process packet_i
    if packet has type 1
        lock(lock_i)
        add packet_i to list_1
        unlock(lock_i)
        cseg_2
    else
        lock(lock_i)
        add packet_i to list_2
        unlock(lock_i)
        cseg_3
end loop
```

*Figure 6.1. Example program to show code segments*

With such a division of code segments, a program can be abstracted as connected code segments. The idea is similar to a control flow graph, but we use code segments instead of basic blocks.
6.1.2 Predicting lock contention

We represent each code segment as a node in a queueing network. Each local computation segment is represented as an infinite-server node and each lock segments as a single-server node. Such a representation reflects the nature of the local computation and lock segments, respectively. Since local computation segments are assumed to contain no synchronization, they can be executed in parallel with each other. An infinite-server node captures this behavior. A single-server node has the property of only serving one job at the same time, making it ideal to describe the behavior of a lock segment. The whole program can then be described as a closed queueing network. To construct the queueing network, we must supply the following parameters

- number of jobs $K$, which is the total number of running threads.
- routing matrix $r$ where $r_{i,j}$ estimates the probability of a thread executing code segment $cseg_j$ after executing code segment $cseg_i$.
- service time at each node, which is the average execution time of the corresponding code segment. We assume the service time is exponentially distributed.

Consider the example method in Figure 6.1, the routing matrix would be

$$
\begin{pmatrix}
0 & p & 1-p \\
1 & 0 & 0 \\
1 & 0 & 0
\end{pmatrix}
$$

assuming the fraction of type 1 packets is $p$. The resulting queueing network is shown in Figure 6.2.

Let $node_1$, $node_2$, $node_3$ denote the nodes for $cseg_1$, $cseg_2$ and $cseg_3$. The service time at $node_1$ is the time of processing a packet (assuming it is exponentially distributed), at $node_2$ is the time to add a packet to list$_1$ and at
node_3 is the time to add a packet to list_2. We assume the service times are exponentially distributed with the average service time as the mean. By instantiating the queueing network with the parameters and solving the queueing network with standard methods (we use mean value analysis), we can estimate the average waiting time at node_2 and node_3, which can be interpreted as the lock contention of lock_1 and lock_2. For example, with p = 0.4, K = 64 and the service time at the three nodes being 3 milliseconds, 1 millisecond and 2 milliseconds, the predicted lock contention at node_2 (accessing lock_2) is 120 milliseconds.

This example shows that we can estimate the lock contention of a parallel program with any number of threads/cores as long as we can extract the parameters needed for the queueing network.

The input parameters to our model can be obtained by profiling a single-threaded run of the target program. Each lock node’s service time can be obtained by recording the lock and unlock time stamps of each lock and taking the average time between the two time stamps. For each local computation node, its service time is calculated as the average time between the previous unlock and the next lock time stamps. The routing matrix can be generated by keeping track of the order of accessing lock segments and the frequencies of executing one lock segment after another. Suppose lock segment cseg_i is followed by a set of segments cseg_1, cseg_2, ... cseg_n for nfollow_1, nfollow_2, ... nfollow_n times, the probability of visiting any node_j (1 ≤ j ≤ n) after visiting node_i is

\[ r_{i,j} = \frac{n\text{follow}_j}{\sum_{1 \leq k \leq n} n\text{follow}_k} \]

Evaluation

We evaluated this model on a set of micro-benchmarks with varying lock holding times and local computation times, and benchmark dedup from the PARSEC benchmark suite [7]. The performance metric of interest is the average lock contention over all threads. We compared the measured and modeled performance metric. For the micro-benchmarks, the average relative error of the model is less than 5%. For benchmark dedup, which includes two parallel pipelines with contended locks, the relative error of our model is 8.46% and 15.17%, for the two pipeline stages when the lock is contended when running the benchmark with 5 – 8 threads in each stage.

Our method has a few limitations. One major limitation is that it assumes the lock holding times do not increase with more threads/cores. This assumption is not necessarily true for locks that protect heavily accessed shared data. Since cores are likely to write to shared data, this may cause cache coherence misses and leads to increased lock holding times with more cores. This in-
creased lock holding times can be estimated with the model that predicts the number of cache coherence misses in Paper II.

Related work

Gilberg [19] described how a queueing network can be constructed to model the behavior of a multi-threaded system with locks. Obtained results were not validated against measurements. Bjorkman and Gunningberg [8] developed queueing network models to predict performance of multiprocessor implementation of two communication protocol stacks, and obtain predictions with less than 10% error. Cui et al [15] observed that such models do not consider effects of contention for hardware and cache protocol resources as the number of cores increases, and hence cannot model the effects of decreasing overall performance, as has been observed for spin locks, when the number of threads and cores increases beyond some thresholds. They developed a discrete event simulator which takes such effects into account.

6.2 Predicting lock contention of delegation locks

If lock contention is a scalability bottleneck when the number of threads increases, there are several ways to reduce the lock contention, one of which is to replace the current lock implementation with another one. It is difficult to estimate if this method can reduce the lock contention before deploying such a replacement.

If the replacing lock is a non-delegation lock, we can use the method discussed in Section 6.1.2 to predict the lock contention with the new lock. However, for delegation locks, this method would not work since different threads may have different behaviors at each lock node. For example, in flat-combining, the combiner thread is responsible for combining and executing all operations; in queue delegation locks, the helper thread executes all critical sections. While such an asymmetric scheme for lock accesses brings performance benefits, we can not simply use the model in Section 6.1.2 to describe the behavior of delegation locks because it does not account for the different behavior of different threads. Instead, we propose the following model for the queue delegation lock. This model can also be extended to model flat combing.

- Two classes of jobs: a helper job and non-helper jobs
- Jobs of both classes have the same routing matrix
- Service time of code segments: since the helper thread may continue executing other threads’ critical sections after finishing its own, we need to include the extra time of execution, whose length is proportional to the number of contended non-helper threads. Since only the helper thread experiences such a delay, we model it by increasing the service time of
the helper jobs at the next infinite-server node after the lock node. For
the non-helper threads, their service time at each each node is simply the
size of the corresponding code segment.
Since the number of contended non-helper jobs is unknown, the resulting
queueing network can not be solved with standard methods. Instead, we solve
it with an iterative fixpoint approach, which aims to reach a stable point at
which the number of contended threads stays the same in the previous and
current iteration. The number of contended threads at each lock node is ini-
tially set to 0. By solving the queueing network, a new value of number of
contended threads at each node is obtained, which is used in the next iteration,
and so on until convergence.

Evaluation
We evaluated the accuracy of the model with a set of synthetic benchmarks by
comparing the predicted lock contention with the measured lock contention
which is averaged over all threads. Our model has an average relative error
of 24% among all 3500 configurations of the benchmark. It is less accurate
(30 – 40% relative error) when the critical sections are small (shorter than
1000 ns). The relative error drops to 14% for 1950 configurations where the
critical section is longer than 1000 ns. As a case study, we analyzed the lock
performance of the kyotocabinet benchmark [28], our model is able to predict
the lock contention with an average of 18% relative error over 2 – 32 threads.
7. Conclusion

In this thesis, we addressed the challenge of developing techniques for predicting performance metrics of programs on multi-core platforms. We developed models for estimating performance, which are based on inputs that can be collected by low-cost software profiling.

*Cache performance*

Caches can significantly reduce memory access latency and save energy, but this benefit highly depends on both the program and cache configuration. Cache miss ratio is an important performance metric. While previous works have developed techniques for estimating the cache miss ratios under different configurations [47] [18], they either handle only a limited range of cache configurations or require an expensive-to-obtain input. Paper I presents a modeling framework based on Markov chains to estimate cache miss ratios under a variety of cache configurations with different cache size, associativity and replacement policy. Compared to previous methods, this work proposes a new generic format for cache performance predictions and only uses the easy-to-collect reuse distance distribution as input. We evaluated the accuracy of our models by comparing its predicted cache miss ratios to the cache miss ratios obtained from cache simulation. The average absolute errors of our models are 0.72%, 0.93% and 0.98% for caches with LRU, PLRU and bit-PLRU replacement policies.

Data sharing may introduce cache coherence misses on multi-core systems. In Paper II, we presented three models to predict the number of cache coherence misses for a multi-threaded program on multi-core systems. The models build on the observation that the occurrence of a coherence miss on a core is caused by another core writing to shared data interleaving with the reuse of the same shared data. We evaluated the accuracy of our models by combining its predicted number of coherence misses with the estimated number of cold and capacity misses using existing methods, and comparing the total number of cache misses with the measured number of cache misses obtained from hardware performance counters. The average relative errors of our three models (uniform, phased and symmetric) are 5.8%, 8.02% and 5.4% respectively. With these predicted number of cache coherence misses, programmers are able to estimate the cost of data sharing on multi-core systems.

*Lock contention*

We presented a queueing network model for predicting the lock contention of multi-threaded programs on multi-core systems in Paper III. Compared to
previous work, the evaluation of this work is done on real hardware instead of simulations. We evaluated this model on a set of micro-benchmarks with varying lock holding times and local computation times, and benchmark dedup from the PARSEC benchmark suite. Our model can accurately predict lock contention for programs with simple lock access patterns. The average relative error of our predicted lock contention compared to the measured lock contention is less than 5% for the micro-benchmarks. The error is larger with benchmark dedup, which is 8.46% and 15.17% for the contended locks used by threads in two pipeline stages.

Compared to non-delegation locks, delegation locks can make better use of the caches on multi-core systems and generates little coherence traffic. In Paper IV, we presented a model to predict the lock contention if the current non-delegation locks is replaced by a QD lock, which is the state of the art delegation lock. This model can be used by programmers to evaluate the lock contention before adopting a delegation lock. We evaluated our model by comparing its predicted lock contention with the measured average lock contention over all threads on a set of synthetic benchmarks with 3500 configurations and a database benchmark (kyotocabinet). The average relative prediction error of our model is 24% over all 3500 configurations. For kyotocabinet benchmark, our model is able to predict the lock contention with an average error of 18% over 2 – 32 threads.

Future work

In Paper I, the proposed model for estimating the cache miss ratio in general has an absolute error below 1% compared to the cache miss ratio obtained from cache simulation. However, our method for estimating set reuse distance distribution from reuse distance distribution introduces 2% error. Future work is to improve the accuracy of this method by looking into how set reuse distance distribution is influenced by the set mapping function of the cache.

An assumption of the model for predicting lock contention in Paper III is that the lock holding times do not change with different numbers of cores. As observed in Paper II, this assumption often does not hold since the number of cache coherence misses in critical sections is likely to increase with more cores, causing the lock holding times to increase. Potential future work is to first apply the models for predicting the number of cache coherence misses in Paper II to estimate the lock holding times with different numbers of cores, then use these estimated lock holding times to predict the lock contention using the model proposed in Paper III. This extension would make the model for predicting lock contention more applicable to realistic programs.

One limitation of our model for predicting lock contention is that it does not consider the overhead of the operating system scheduler when the lock is contended. Such an overhead can play a significant role in lock contention for
some lock libraries, such as the Pthread mutex lock. It remains unclear to us how the current model can take this overhead into account for now. It would be worthwhile to investigate into this challenge since Pthread mutex locks are widely used in multi-threaded programs.

A general problem of evaluating our models of predicting lock contention in Paper III and Paper IV is the lack of standard benchmark suite with lock contention. This is partly due to the fact that widely used benchmark suites have been optimized to reduce lock contention. As future work, we could write benchmarks with realistic lock access patterns and evaluate the accuracy of our models with them.
8. Sammanfattning på svenska

Numera har datorer blivit en oumbärlig del av våra dagliga liv. Vi vill att de ska göra våra liv enklare och mer behagliga genom att uppfylla våra behov och krav. Sådana krav kan relateras till ett systems funktionalitet, som till exempel att en nerladdningsprogram skall ge en ljudsignal när nerladdningen är färdig, att en musiktjänst skall komma ihåg vilka låtar som har spelats, att en krockkudde ska lösa ut vid en kollision eller att ett teve-spelet skall tillåta att man fryser spelat och återupptar det senare. Krav kan också vara prestandarelaterade, som till exempel att en nerladdningsprogram skall kunna ske med minst 1Gbit/s, att en krockkudde ska lösa ut inom 100 millisekunder efter en kollisionsdetektering, att musiken ska spelas utan avbrott, eller att teve-spelet skall ha en uppdateringsfrekvens på 60 bilder i sekunden. Dessa prestandakrav kan vara lika viktiga som de funktionella kraven. Om de inte uppfylls kan användarupplevelsen i bästa fall försämras och i värsta fall kan programmet vara totalt värdelöst.

Sedan 1950 talet har datorer blivit allt mer kraftfulla för att kunna exekvera program snabbare och mer effektivt. Detta har åstadkommits genom en rad olika förbättringar: processorhastigheten har ökat, accesstiden till minne har kortats, instruktionsschemaläggningen har förbättrats, etc. Man skulle kunna tro att med dagens kraftfulla datorer skulle det vara enkelt att uppnå maximal prestanda för alla applikationer. Tyvärr är detta inte fallet. Det har visat sig vara mycket svårt för ett program att utnyttja ett datorsystems maximala prestanda.

Sedan 2005 har huvudfrekvensen hos de snabbaste processorkärnorna inte ökat särskilt mycket på grund av att det inte är kostnadseffektivt att kyla processor som drar mycket mer än 100 watt. Processortillverkarna har i stället börjat öka antalet processorkärnor på varje chip för att förbättra prestandan samtidigt som effektförbrukningen hålls relativt låg. Flerkärniga processor har blivit mycket populära, och det är till och med vanligt att mobiltelefoner har fyra eller fler processorkärnor.

Givet hur viktigt det är med hög prestanda och hur svårt det är för moderna program att åstadkomma just detta så behöver programmerarna verkyg för att kunna förstå hur deras program utnyttjar datorsystemets resurser. Till exempel, använder programmet alla tillgängliga processorkärnor i systemet? Finns data i cachen när programmet behöver det? Hur mycket tid ägnar programmet åt synkronisering?

I denna avhandling presenterar vi tekniker som hjälper programmerare att analysera sina program’s prestanda. Vi fokuserar på två fundamentala aspekter
av prestandan hos flerkärniga system, nämligen cache-prestanda och synkroniseringskostnad.


References


A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title “Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology”.)