UPPSALA UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1000

Profiling Methods for Memory Centric Software Performance Analysis

DAVID EKLÖV

ACTA UNIVERSITATIS UPSALIENSIS UPPSALA 2012

ISSN 1651-6214
ISBN 978-91-554-8541-2
urn:nbn:se:uu:diva-182594

Dissertation presented at Uppsala University to be publicly examined in Informationsteknologiskt centrum, Polacksbacken, Pol/2446, Lägerhyddsvägen 1, Uppsala, Friday, December 21, 2012 at 13:00 for the degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract

Eklöv, D. 2012. Profiling Methods for Memory Centric Software Performance Analysis. Acta Universitatis Upsaliensis. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1000. 51 pp. Uppsala. ISBN 978-91-554-8541-2.

To reduce latency and increase bandwidth to memory, modern microprocessors are often designed with deep memory hierarchies including several levels of caches. For such microprocessors, both the latency and the bandwidth to off-chip memory are typically about two orders of magnitude worse than the latency and bandwidth to the fastest on-chip cache. Consequently, the performance of many applications is largely determined by how well they utilize the caches and bandwidths in the memory hierarchy. For such applications, there are two principal approaches to improve performance: optimize the memory hierarchy and optimize the software. In both cases, it is important to both qualitatively and quantitatively understand how the software utilizes and interacts with the resources (e.g., cache and bandwidths) in the memory hierarchy.

This thesis presents several novel profiling methods for memory-centric software performance analysis. The goal of these profiling methods is to provide general, high-level, quantitative information describing how the profiled applications utilize the resources in the memory hierarchy, and thereby help software and hardware developers identify opportunities for memory related hardware and software optimizations. For such techniques to be broadly applicable the data collection should have minimal impact on the profiled application, while not being dependent on custom hardware and/or operating system extensions. Furthermore, the resulting profiling information should be accurate and easy to interpret.

While several use cases are presented, the main focus of this thesis is the design and evaluation of the core profiling methods. These core profiling methods measure and/or estimate how high-level performance metrics, such as miss-and fetch ratio; off-chip bandwidth demand; and execution rate are affected by the amount of resources the profiled applications receive. This thesis shows that such high-level profiling information can be accurately obtained with very little impact on the profiled applications and without requiring costly simulations or custom hardware support.

David Eklöv, Uppsala University, Department of Information Technology, Computer Systems, Box 337, SE-751 05 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-554-8541-2
urn:nbn:se:uu:diva-182594 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-182594)

To Ann-Sofie

List of papers

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

I David Eklov and Erik Hagersten. *StatStack: Efficient Modeling of LRU Caches*. In Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), White Plains, NY, USA, March 2010

II David Eklov, David Black-Schaffer and Erik Hagersten. *Fast Modeling of Cache Contention in Multicore Systems*. In Proceedings of the the 6th International Conference on High Performance and Embedded Architecture and Compilation (HiPEAC), Heraklion, Crete, Greece, January 2011

III David Eklov, Nikos Nikoleris, David Black-Schaffer and Erik Hagersten. *Cache Pirating: Measuring the Curse of the Shared Cache*. In Proceeding of the 40th International Conference on Parallel Processing (ICPP), Taipei, Taiwan, September 2011

IV David Eklov, Nikos Nikoleris, David Black-Schaffer and Erik Hagersten. *Quantitative Characterization of Memory Contention*. Technical Report 2012-029, Department of Information Technology, Uppsala University, Sweden, October 2012. Submitted for publication.

V David Eklov, Nikos Nikoleris and Erik Hagersten. *A Profiling Method for Analyzing Scalability Bottlenecks on Multicores*. Technical Report 2012-030, Department of Information Technology, Uppsala University, Sweden, October 2012. Submitted for publication.

Reprints were made with permission from the publishers. All papers are verbatim copies of the original publications but are reformatted to the one column format of this book.

Comments on my Participation

I I am the principal author and principal investigator.

II I am the principal author and principal investigator.

III I am the principal author and principal investigator. Nikos Nikoleris contributed to the discussions and provided reference data.

IV I am the principal author and principal investigator. Nikos Nikoleris contributed to the discussions.

V I am the principal author and principal investigator. Nikos Nikoleris contributed to the discussions, wrote the initial version of the instrumentation framework and helped porting the OpenMP benchmarks to Pthreads.

Other Publications

Conference Papers

- Andreas Sandberg, David Eklov and Erik Hagersten. *Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses*, In Proceedings of Supercomputing (SC), New Orleans, LA, USA, November 2010
- Andreas Sembrant, David Eklov and Erik Hagersten. *Efficient Software-based Online Phase Classification*, In Proceeding of the International Symposium on Workload Characterization (IISWC), Austin, TX, USA, November 2011,

Book Chapters

- Erik Hagersten, David Eklov and David Black-Schaffer. *Efficient Cache Modeling with Sparse Data*, Chapter in "Processor and System-on-Chip Simulation", editors Olivier Temam and Rainer Leupes. Springer, 2010

Extended Abstracts

- David Eklov, David Black-Schaffer and Erik Hagersten. *StatCC: A Statistical Cache Contention Model*, In the Proceedings of 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), Vienna, Austria, September 2010
- David Eklov, Nikos Nikoleris, David Black-Schaffer and Erik Hagersten. *Bandwidth Bandit: Understanding memory Contention*, In the Proceedings of 21th International Conference on Parallel Architectures and Compilation Techniques (PACT), Minneapolis, MN, September 2012

Contents

1 Introduction ... 11
1.1 Contributions ... 12
1.1.1 Statstack ... 12
1.1.2 StatCC ... 12
1.1.3 Cache Pirating ... 12
1.1.4 Bandwidth Bandit ... 13
1.1.5 Speedup Stacks ... 13
1.2 Thesis Outline ... 13

2 Background ... 15
2.1 Memory Hierarchies ... 15
2.2 Data Locality ... 16
2.3 Performance Counters ... 17

3 Statstack and StatCC ... 18
3.1 Input Data: Reuse Distance Distribution ... 18
3.2 Output Data: Stack Distance Distribution ... 20
3.3 Cache Misses ... 22
3.3.1 Compulsory Misses ... 22
3.3.2 Capacity Misses ... 22
3.3.3 Conflict Misses ... 23
3.3.4 Misses due to Cache Sharing ... 23
3.4 Miss Ratio Curves ... 24

4 The Pirate and the Bandit ... 27
4.1 Cache Pirate ... 29
4.1.1 Case Study ... 31
4.2 Bandwidth Bandit ... 32
4.2.1 Case Study ... 36

5 Speedup Stacks ... 38
5.1 Speedup Stacks ... 39
5.2 Case Study 1: Single-threaded applications ... 40
5.3 Case Study 2: Multi-threaded applications ... 42

6 Summary and Conclusions ... 44

7 Introduction in Swedish ... 45

8 Acknowledgement 48

References 49

1. Introduction

The long latency and limited bandwidth to off-chip memory have been identified as two potential bottlenecks of present and future computer systems [25, 33]. To reduce latency and increase bandwidth to memory, modern microprocessors are often designed with deep memory hierarchies including several levels of caches. For such microprocessors, both the latency and the bandwidth to off-chip memory are typically about two orders of magnitude worse than the latency and bandwidth to the level-one cache. Consequently, the performance of many applications is largely determined by how well they utilize the caches and bandwidths in the memory hierarchy.

For such applications, there are two principal approaches to improve performance: optimize the memory hierarchy and optimize the software. Hardware developers can, for example, increase cache size and bandwidths or change the cache organization in order to better match the requirements of the target workloads. Software developers, on the other hand, might, for example, use different data structures and/or reorganize the memory access pattern to match the memory hierarchy on the target computer system. In order to be successful, it is important, for both hardware and software developers, to qualitatively and quantitatively understand how the software interacts with the memory hierarchy.

To this end, this thesis presents the design and evaluation of several novel profiling methods for memory-hierarchy-centric software performance analysis. The goal of these profiling methods is to provide general, high-level, quantitative information describing how the profiled applications utilize the resources (e.g., cache and bandwidths) in the memory hierarchy, and thereby help software and hardware developers to identify opportunities for, and assess the benefit of, memory related hardware and software optimizations. For such techniques to be broadly applicable the data collection should have minimal impact on the profiled application, while not being dependent on custom hardware and/or operating system extensions. Furthermore, we argue that for software developers it is often important to obtain profiling information that captures the software's behavior on real hardware.

While several use cases are presented, the main focus of this thesis is not on specific applications of these profiling methods (e.g., compiler optimizations, process schedulers and thread placement algorithms). Instead, the focus is on the core profiling technologies, their implementation and evaluation, and how to interpret the resulting profiling information in a general context. These core profiling methods measure and/or estimate how high-level performance metrics, such as miss- and fetch ratio; off-chip bandwidth demand; and execution rate are affected by the amount of resources profiled applications receive. This thesis shows that such high-level profiling information can be accurately obtained on real hardware with very little impact on the profiled application and without requiring costly simulations or custom hardware support.

1.1 Contributions

1.1.1 Statstack

Statstack (Paper I) is a profiling method that captures the profiled application's Miss Ratio Curve (MRC). The MRC reports the application's cache miss ratio as a function of its available cache capacity and thereby captures the profiled application's data locality. There are many ways in which MRC can be captured, e.g., full system simulations (e.g., [3, 21]), binary instrumentation (e.g., [20, 24]) and stack distance based methods (e.g., [11, 19, 30, 34]). All these methods incur two main costs. The cost of capturing the profiling information and the cost of (post) processing it. Statstack reduces both these costs by employing statistical sampling techniques.

1.1.2 StatCC

In order to improve the utilization of the available cache capacities, many multi-cores share the last-level cache among its cores. This enables programs/threads that have higher demands for cache capacity to use more of the shared cache than other co-running programs/threads that demands less cache capacity. This, however, can adversely increase the number of cache misses of the co-running programs/threads (compared to when they get to use equal shares of the cache capacity). StatCC (Paper II) enables us to estimate the number of additional misses incurred due to cache sharing for co-running instances of single-threaded applications. This method is based on Statstack and therefore enjoys the same low cost of collecting profiling information.

1.1.3 Cache Pirating

Cache Pirating (Paper III) is a method for obtaining any performance metrics available through hardware performance counters, such as miss- and fetch-ratios, off-chip bandwidth demand, and execution rate, as a function of the profiled application's available cache capacity. This data is obtained while the profiled application runs on a real, commodity multi-core. As a result, Cache Pirating has the following benefits. First, the profiling data represents the profiled application's behaviour on real multi-core, as opposed to a simulated multi-core, including out-of-order execution, memory level parallelism and hardware prefetching. Second, the profiling overhead is low compared to simulation and instrumentation-based alternatives.

We present several examples of how to interpret the data captured using Cache Pirating, and a case study in which we show how Cache Pirating can be used to understand and predict how the memory hierarchy limits the throughput when co-running multiple instances of a single-threaded application.

1.1.4 Bandwidth Bandit

While Cache Pirating can measure an application's off-chip bandwidth demand, it cannot be used to determine how the application is affected the by off-chip bandwidth limitations that can arise when multiple programs/threads share the off-chip memory bandwidth. The Bandwidth Bandit (Paper IV) is a method for obtaining any performance metrics available through hardware performance counters as a function of the profiled application's available off-chip memory bandwidth. This data enables us to determine how the profile application is affected by off-chip memory bandwidth limitations. Similarly to Cache Pirating, this data is obtained while the profiled application runs on a real, commodity multi-core, and therefore has the same benefits as Cache Pirating.

We present several examples of how to interpret the data captured by the Bandwidth Bandit. Furthermore we present a case study in which we show how the captured data can be used to predict how off-chip memory bandwidth limitations impact the throughput when co-running multiple instances of a single-threaded application, which cannot be done based solely on data captured with Cache Pirating.

1.1.5 Speedup Stacks

A key property of multi-threaded programs is how their execution times scale when increasing the number of threads. However, there are several factors that can limit the scalability of multi-threaded programs. A speedup stack reports how much each of these scalability bottlenecks limits the program's speedup. This thesis presents a method for obtaining speedup stacks for both multiple co-running instances of single-threaded applications and for multi-threaded applications (Paper V). It captures speedup stacks while the profiled program runs on a real, commodity multi-core, and therefore has similar benefits as Cache Pirating and the Bandwidth Bandit.

1.2 Thesis Outline

The remainder of this thesis is organized as follows. First, we briefly discuss how memory hierarchies are organized in modern general purpose multi-core processors and highlight some of their performance implications, including a discussion about the data locality principle (Chapter 2). Then, we introduce Statstack, our main contribution in the area of data locality analysis (Chapter 3). This chapter includes a discussion about the different types of cache misses and, importantly, which of these types of misses Statstack estimates. We then introduce Cache Pirating and the Bandwidth Bandit (Chapter 4). The main focus of this chapter is on how interpret the data obtained using the methods. In addition, we also present several case studies in which we use the Cache Pirate and the Bandwidth Bandit to predict and explain the throughput of co-running applications. Finally, we introduce a novel profiling method for obtaining speedup stacks on real, commodity multi-cores (Chapter 5).

2. Background

2.1 Memory Hierarchies

The performance of many software applications is largely determined by the speed at which the memory system can deliver data. There are many different memory technologies with different speed characteristics that can be used in a memory system. Ideally, one would want memory systems to make exclusive use of the fastest memory technology available. However, as is often the case, there is a trade-off between speed, capacity and cost. Given the large memory demands of today's applications, it is generally not feasible for a memory system to exclusively use the fastest available memory technology. The conventional solution is to build a hierarchical memory system. Most applications access a few, relatively small parts of their data more frequently than others. Hierarchical memory systems therefore use small but fast memories to hold the most frequently used data, while the less frequently used data is kept in large but slow memories. This way, most of the data requests will be serviced by the small, fast memories while the large, slow memories ensure that memory system can accommodate large data demands.

Figure 2.1 shows a typical memory hierarchy found in modern multi-core servers and desktop processors (similar to those of Intel's Nehalem and Sandy Bridge architectures). It has three levels of caches, L1, L2 and L3, which are are small, fast memories, residing on the same chip as the processor cores. Each core has a private L1 and L2 cache, while the L3 cache is shared among the cores. Each consecutive cache level is larger, but slower, than the previous, lower level. The caches are managed in hardware which uses heuristics to identify and store the most frequently used data in the cache memories. Since caches are managed by hardware, they are functionally transparent to software. Higher up in the memory hierarchy is the main memory, which consists of relatively large, but slow, off-chip (i.e., external) memory modules. The cores communicate with the main memory via an on-chip (i.e., integrated) memory controller. The memory controller has three independent memory channels, each connecting up to three memory modules. Contrary to the caches, the main memory is managed by software (often the operating system). That is, the software has full control over what data is stored in the main memory. At the top of the memory hierarchy are the hard drives. They too are managed by software, typically using a file system. Compared to main memory, traditional hard drives are much slower, but also much cheaper (per bit of storage) than main memory. Typical server and desktop computers therefore have substantially larger hard drive capacities than main memory capacities.

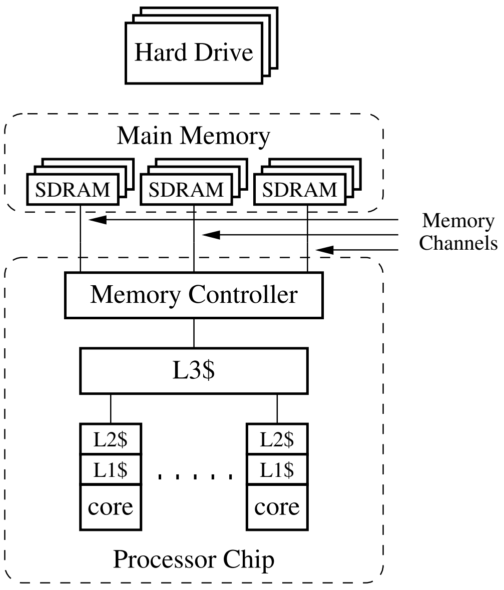


Figure 2.1. Memory hierarchy of a typical, modern, mulit-core processors.

2.2 Data Locality

Central to the success of caches is their ability to exploit applications' *data locality*. Data residing in memory is referred to by its memory address (i.e., its location in main memory). The data locality principle [9, 10] says that recently accessed memory locations, and nearby memory locations, are more likely to be accessed in the near future, than other, non-related memory locations. As defined above, data locality can be broken up into two basic types: *spatial-* and *temporal* locality. Spatial locality refers to the phenomena that a memory accesses to a certain memory location is often closely followed by memory accesses to nearby memory locations. This is a result of the common programming practice of storing related data items at nearby memory locations, for example, using programming constructs such as classes and/or arrays. Temporal locality, on the other hand, refers to the phenomena that recently accessed memory locations are likely to be accessed again in the near future.

Caches exploit applications' data locality the following main ways. The basic unit with which an application reads and writes data is called a *data word*. However, in the event of a cache miss, the cache controller fetches and installs a whole cache block, containing the requested data words as well as several of its immediate neighbors. This effectively exploits an application's spatial locality by potentially turning accesses, performed in the near future, to the additionally fetched data words into cache hits. Temporal locality, on the other hand, is exploited by keeping the most recently accessed cache blocks in the cache, which, according the principle of temporal locality, are likely to be accessed again in the near future. This is the responsibility of the cache replacement policy. For example, the Least Recently Used (LRU) replacement policy attempts to evict the least recently used cache block, thereby ensuring that the most recently used cache blocks remains in the cache.

It is important to note that the data locality principle is merely the observation that many commonly used programming patterns often have an inherit data locality. It does not grantee that all applications have large degrees of data locality. However, as the data locality of an application has large impact on the effectiveness of caches, and consequently the application's performance, it is highly desirable that applications ample good data locality.

2.3 Performance Counters

In order help programmers to optimize their applications, most processors implement a set of hardware performance counters that can be programmed to count the occurrence of various events during the execution of an application. For example, they can be programmed to count the number of memory requests serviced by the different levels of the memory hierarchy. These event counts can be used to determine whether an application suffers from memory hierarchy related performance issues. Sometimes, they can even be used to identify the specific instructions and/or data structures that are causing the memory issues. For expert programmers, this information can be enough to figure out how to resolve the memory issues, which typically involve using alternative data structures and/or reorganizing the computations. This, however, might involve fair amount of trial and error and can be a both a tedious and time consuming process. Since programmer productivity is important for keeping development costs low, methods and tools that simplify the above optimization process is therefore very valuable.

3. Statstack and StatCC

Statstack, presented in Paper I, is a profiling method for capturing applications' *miss ratio curves* (MRCs) [22] for fully-associative caches with least recently used replacement policies. The information captured by Statstack is fundamental to data locality analysis. Statstack is inspired by StatCache [4, 5] – a fast cache model for caches with random replacement. Specifically, Statstack uses the same input data as StatCache, and can therefore use the low-overhead profiling method of StatCache to collect profiling data.

Statstack can, in some respects, be thought of as an alternative to cache simulators. A simple trace driven cache simulator takes as input a *memory access trace*, i.e. the sequence of memory addresses referenced by an application during its execution. Often, the result of the simulation is the application's cache hit ratio[1] for the simulated cache size (or potentially several different cache sizes). However, simulating a cache can be expensive. It typically has two main costs. First, the cost of capturing the memory access trace. This can be done in many ways. For example using dynamic instrumentation frameworks such as PIN [20] and Valgrind [24] or using system simulators such as Simics [21] and Simplescalar [3]. As a memory access trace contains all memory addresses accessed by the profiled application, capturing the trace requires each memory instruction (i.e., load and store instructions) to be instrumented. As a result, the instrumentation overhead can be significant. The second cost is that of running the actual cache simulation. For each simulated memory access, the simulator has to the following: 1) Determine whether the accessed data is in the (simulated) cache and record the access as a hit or a miss accordingly. 2) Updated the state required by the replacement policy, and in the case of a cache miss, simulate the replacement policy in order to determine which cache block to evict. While each such sequence of operations is not very expensive, performing them for each memory access in the address trace can add up to a significant cost. Statstack aims at reducing both of these costs.

3.1 Input Data: Reuse Distance Distribution

Figure 3.1 gives a high-level overview of Statstack. Unlike a trace-driven cache simulator, Statstack does not require a complete memory access trace.

[1]In this work, we define hit/miss ratio as the number of cache hits/misses divided by the number of memory accesses.

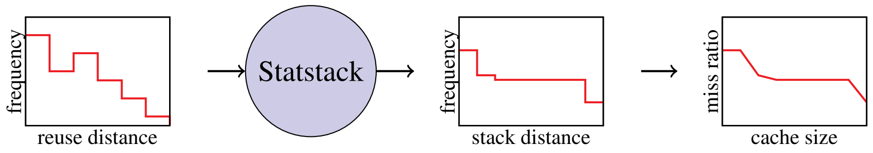


Figure 3.1. Statstack flowchart. The input to Statstack is a sparse reuse distance distribution (far left). As an intermediate step, Statstack produces a stack distance distribution (second from right), and, finally, it produces a miss ratio curve (far right).

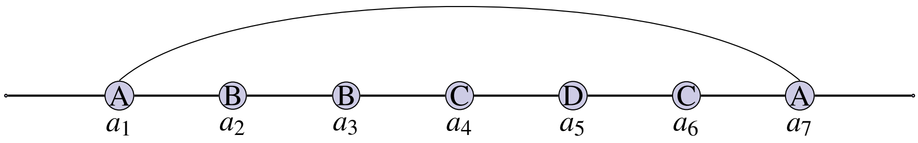


Figure 3.2. Subsequence of a memory access trace. The circles on the horizontal time axis represent memory accesses. The memory accesses a_1 and a_7 both access cache block A. The *reuse distance* of a_1 is therefore five $((7-1)-1)$. The five memory accesses executed between memory access a_1 and a_7 reference three unique cache blocks, namley B, C and D. The *stack distance* of a_1 is therefore three.

Instead, its input is the profiled application's *reuse distance distribution*. A *reuse distance* is the number of memory accesses executed between two consecutive memory accesses to the same cache block. This is illustrated in Figure 3.2. Here, a_1 and a_7 are two consecutive memory accesses to cache block A. The reuse distance of memory accesses a_1 is equal to five since there are five memory accesses executed between a_1 and a_7. Furthermore, suppose that a_7 is the last memory access to reference cache block A. In this case, the reuse distance of a_7 is defined to be infinite. Figure 3.3 shows an example reuse distance distribution. An application's reuse distance distribution is simply the distribution of its memory accesses' reuse distances.

The nature of the reuse distance is such that individual reuse distance can be easily and efficiently measured using hardware performance counters and hardware memory watch points. Additionally, a reuse distance distribution can be accurately estimated based on a sparse sample of the profiled application's memory access. These two factors combined allow for very low cost data capturing. Berg et al. [5] presents a accurate reuse distance sampler that measures the reuse distances of randomly selected memory accesses. Their sampler use hardware performance counters readily available on most commodity processor. As such it does not require any custom hardware support. On average, the sampled application's is slowed down by 40%. In order to further reduce the overhead of Berg's original sampler, Sembrant et al. [28] extended it with an online program phase detection mechanism [29]. They rely on the observation that recurring program phases typically have very sim-

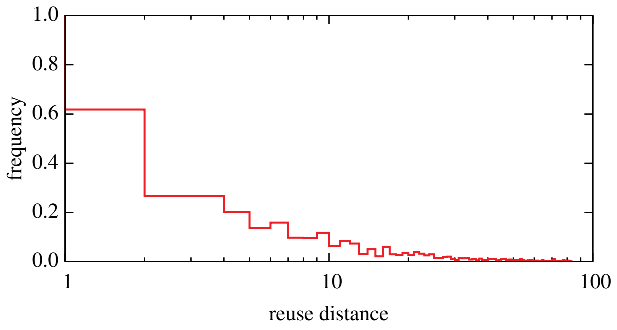


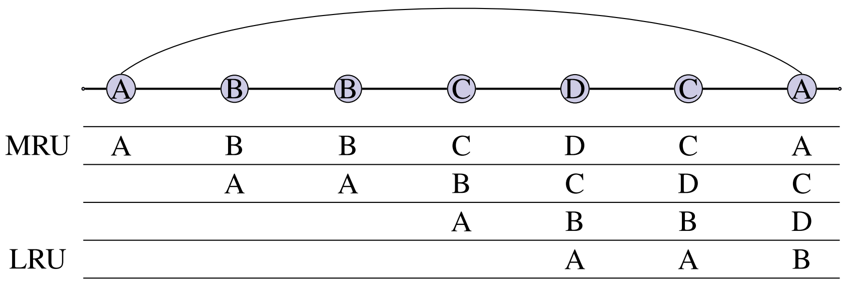
Figure 3.3. An example reuse distance distribution. (Note the log scale on the horizontal axis.)

ilar reuse distance distribution, and only collect reuse distance distributions of the first occurrence of each unique program phase. This reduces the slowdown of the sampled applications to 20%, on average.

Under the fairly loose definition of temporal locality outlined in Chapter 2, the reuse distance can be viewed as a direct measure of temporal locality. The way to think about this is as follows. The cache state is only updated in the event of a memory accesses. From the point of view of the cache, time can therefore be measured in executed memory accesses. The reuse distance measures the time between two accesses to the same cache block and thereby measures temporal data locality. However, while being a measure of temporal locality, the reuse distance does not necessarily indicate whether a previously accessed cache block still remains in the cache when it is accessed for a second time. For random replacement policies, this is instead determined by the number of evictions (i.e., cache misses) that has occurred since the cache block was last accessed [5]. This is not necessarily the same as the reuse distance of the pair of accesses referencing the cache block. For caches with LRU replacement, this is instead determined by the *stack distance*.

3.2 Output Data: Stack Distance Distribution

Statstack is based on a set of mathematical formulas which takes as input the profiled application's reuse distance distribution and computes its *stack distance distribution* for a fully-associative cache. This only requires two passes of relatively inexpensive computations over the reuse distance distribution and is therefore much less costly than full cache simulations. (For details see Paper I.) The *stack distance* [22] is the number of unique cache blocks accessed between two consecutive accesses to the same cache block. This is illustrated in Figure 3.2. Here, the stack distances of a_1 is equal to three, since there are



MRU	A	B	B	C	D	C	A
		A	A	B	C	D	C
				A	B	B	D
LRU					A	A	B

Figure 3.4. LRU Stack. The figure shows the contention of the LRU stack after the execution of the memory accesses on the horizontal time-line.

three unique cache blocks accessed by the memory accesses executed between a_1 and a_7.

The stack distance is closely tied to the LRU replacement policy. Conceptually, the LRU replacement policy maintains the cache lines on a fixed sized stack whose size equals the number of cache lines in the cache. On a cache hit, the accessed cache line is moved to the top of the stack. On a cache miss, the cache block held in the cache line at the bottom of the stack is evicted. The newly fetched cache block is installed in this cache line which is moved to the top of the stack. This is illustrated in Figure 3.4. In this context, the stack distances can be alternatively defined as the accessed cache line's distance from the top of an infinite LRU stack at the time when it is accessed. Memory accesses whose stack distances are less than or equal to the cache size (measured in cache lines), therefore results in cache hits. Otherwise, if their stack distances are less than the cache size, they result in cache misses.

Internally, before computing the stack distance distribution, Statstack first computes a mapping from reuse distances to stack distances. This mapping can then applied as follows: 1) To the reuse distances of a specific memory instruction, thereby producing the stack distances of the instruction. This enables data locality analysis of individual instructions. 2) To the reuse distances of all memory instructions accessing a specific data structure, thereby producing the stack distance distribution of the data structure. This allows for data locality analysis of individual data structures. 3) To the application's entire reuse distance distribution to get the application's stack distance distribution. This flexibility enables Statstack to be used for many different data locality analyses. For example, Statstack have been integrated in Acumem Virtual Performance Expert [14]. Furthermore, we have shown that Statstack can been effectively used to find memory instructions that pollute the caches with data that is not reused while live in the caches [27]. This information was then feed to a compiler optimization pass that automatically reduce the cache pollution.

3.3 Cache Misses

Before discussing how Statstack estimates miss ratio curves, we look at some of the causes of cache misses. A cache miss occurs when a memory request is made to a cache block that is not currently resident in the cache. There are several reasons for why a cache block is not cached and therefore several reasons for why cache misses occur. Hill et al. [17] categorized cache misses into the following three broad categories: *compulsory*, *capacity* and *conflict* misses.

3.3.1 Compulsory Misses

A compulsory cache miss occurs when a cache block is accessed for the first time. The first access to a it must necessarily result in a cache miss, unless the cache block was prefetched. As such, compulsory cache miss are independent of the cache size, i.e., increasing the cache size will not reduce the number of conflict misses. The number of compulsory misses experienced by an application is proportional to its *memory footprint*, i.e., the total amount of memory accessed by the application.

Statstack estimates the number of compulsory cache misses by counting the number of infinite reuse distances in the application's reuse distance distribution. All cache blocks accessed by an application must necessarily be accessed one last time, resulting in an infinite reuse distance. However, as Statstack uses a sampler that only measures the reuse distances of a few randomly selected memory accesses, the number of infinite reuse distances in the reuse distance distribution must be multiplied with the sample period in order to get the number of compulsory cache misses.

3.3.2 Capacity Misses

For fully associative caches there are only two types of misses: compulsory and capacity. Capacity misses, as the name suggests, are due to limited cache capacity. In the event of a cache miss, the cache controller might have to evict a previously installed cache block to make room for the accessed cache block. If the evicted cache block is later accessed (after having been evicted) that memory access will result in a capacity miss. Since it is the responsibility of the cache replacement policy to select which cache blocks to evict, the number of capacity misses experienced by an application is determined by the effectiveness of the replacement policy.

For LRU caches, a capacity miss occurs when the volume of data accessed since the last time the cache block in question was accessed exceeds the cache capacity. Recall that the stack distance is the number of unique cache blocks accessed between two consecutive accesses to the same cache block. Since the volume of data accessed during a given time interval is equal to the number of unique cache blocks accessed during the interval, the stack distance determines whether a memory access results in a capacity miss or not.

3.3.3 Conflict Misses

Conflict misses are due to conflicts between cache blocks mapping into the same the set of a set-associative cache. How to define and measure conflict misses is still an open question [26]. The traditional approach, however, is to simulate a both fully associative and a set-associative cache of the same size, and compare the resulting number of misses. The difference between the two are then the conflict misses [16]. While this a fairly reasonable approach, it has a few draw backs. For example, it is possible that an application has a lower miss ratio for a set associative cache than a fully associative cache. The traditional method of computing the conflict miss ratio would in this case yield a negative conflict miss ration.

Statstack does not handle conflict misses. Most of today's benchmark applications have relatively few conflict misses (compared to capacity misses) for caches with reasonable associativities, so presently this is not a big issue. However, in the future we might see machines with many cores sharing caches of low associativity, resulting in relatively few cache-ways per core and potentially more conflict misses. However, we argue that, for the general-purpose data locality analysis conflict misses are less relevant. Data locality is a property of the applications. However, for a given application, the number of conflict misses are determined by the cache architecture, specifically its associativity, and therefore not a property of the application.

3.3.4 Misses due to Cache Sharing

Cache sharing – i.e., when multiple cores share a cache (e.g., see the level-three cache in Figure 2.1) – can improve the utilization of the shared cache capacity. However, as alluded to above, it can also result in additional cache misses (as compared to when one core gets all the cache capacity). This occurs because the threads running on the cores sharing cache get to use only a fraction of the total shared cache capacity[2]. While this class of misses are not essential for interpreting the data provided by Statstack, this thesis presents method, called StatCC, presented in Paper II, based on Statstack, that estimates these additional cache misses, which we briefly talk about here.

StatCC is a method that accurately predicts the shared cache miss ratios of a set of co-running applications based on their individual reuse distance distributions. This allows StatCC to use the low-overhead samplers presented

[2]Technically, for set-associative caches, each thread competes for and gets to use a fraction of each cache-set. StatCC, however, ignores this and instead models a fully-associative cache. For details see Paper II.

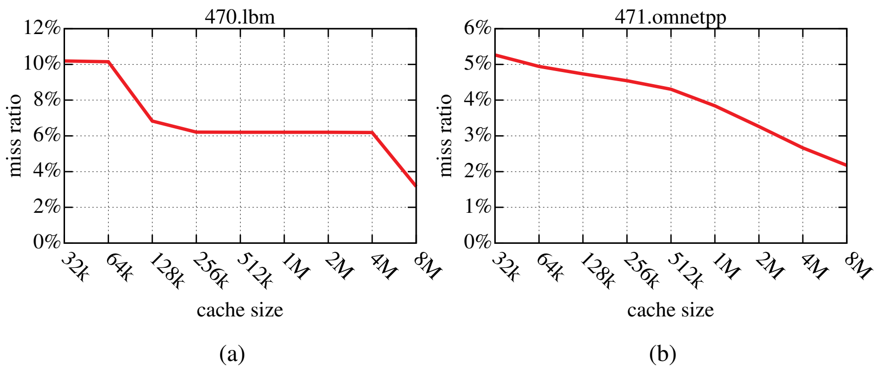


Figure 3.5. Miss ratio curves for 470.lbm (a) and 471.omnetpp (b).

by Berg et al. [5] and Sembrant et al. [28] to collect the required profiling information. StatCC is best described as a two-step process. First, it predicts the shared cache miss ratios of the co-running applications, given that we know their relative CPIs. This is done by transforming the individual reuse distance distributions of the co-running applications into the reuse distance distribution of the interleaved access stream presented to the shared cache. Once we have this combined distribution, we can use Statstack to estimate the shared cache miss ratio. Of course, this method is of little use unless we know the CPIs of the co-scheduled applications to begin with. Under the assumption that we can predict an application's CPI based upon its miss ratio (e.g., using [1, 13, 18, 31]), we can write the following simple equation system, shown here for two co-running applications:

$$\begin{cases} cpi_1 = CPI\left(miss\_ratio_1\left(cpi_1, cpi_2\right)\right) \\ cpi_2 = CPI\left(miss\_ratio_2\left(cpi_1, cpi_2\right)\right). \end{cases} \tag{3.1}$$

This equation system can be readily solved for CPIs of the co-scheduled applications (cpi_1 and cpi_2) by a general purpose equation solver. We then use these CPIs to compute the applications' shared cache miss ratios ($miss\_ratio_i(cpi_1, cpi_2)$).

3.4 Miss Ratio Curves

Once Statstack has computed the application's stack distance distribution, the application's cache miss ratio, for any given range of cache sizes (i.e., the application's MRC), can be easily calculated in a single pass over the stack distance distribution. The miss ratio for a given cache size is simply the fraction of memory accesses whose stack distances is greater than or equal to the cache size (measured in cache lines). This fraction can be computed for all cache sizes in a single pass over the stack distances distribution.

Figure 3.5 show the MRCs for LBM and OMNet++ captured by Statstack. LBM's MRC has two distinct, large knees, at 64kB and 4MB. This indicates that LBM accesses two distinct data sets that require cache sizes of 64kB and 4MB to fit in the cache, respectively. This however is not necessarily the sizes of the data sets. Accesses to these data sets might be interleaved with accesses to other data. These sizes (64kB and 4MB) therefore indicate the cache sizes needed to hold specific data set plus the interleaved data. Furthermore, the flat plateaus to the left of the knees indicate poor data locality. Consider the data set corresponding to the knee at 4MB. Since the MRC is flat between 256kB and 4MB, any additional cache space beyond 256kB will not be utilized unless the cache is larger than 4MB. We note that MRCs with sharp knees and flat plateaus (like LBM's) often indicate regular, sequential access pattern. This observation will be further discussed in Section 4.1. OMNet++'s MRC is significantly different. It gradually drops off. (Notice that the x-axis is on log-scale.) This means that OMNet++ benefits from gradually larger cache sizes, it therefore has better data locality than LBM.

At this point one might ask: Why do we need MRCs when an application's miss ratio can be measured with hardware performance counters? The simple answer is that performance counters only gives us the miss ratios for the caches sizes available on the specific machine on which the measurements are performed. While this information shows how well the application utilizes the caches of this machine, it does not provide much information about the application's data locality. Importantly, in cases where the cache utilization is poor, it does not provide much insight into what is causing the poor utilization. An important example of when the miss ratio provided by performance counters is insufficient is when measuring an application's *working-set*.

The working set of an application can be loosely defined as the smallest cache size for which the application does not experience any capacity misses. The working set is not necessarily the same as the footprint (i.e., the amount of data accessed during an entire execution). For example, consider an application that accesses every element of a large array once. This application has very poor data locality, however since it does not reuses any of its data it will not experience any capacity misses. (It will however experience compulsory misses.) Its working set size is zero, while its footprint equals the size of the array. Given the miss ratio for a specific cache sizes provided by performance counters, we can only determine whether the working set is smaller or larger than this specific cache sizes. Roughly, if the miss ratio is zero, the working set is smaller than the cache size, otherwise it is larger.

When analyzing an application's MRC we can easily find the application's working set. First, consider applications with trivial compulsory miss ratios. For such applications, the working-set sizes is, by definition, equal to the cache size for which the MRC intersects the x-axis, i.e., becomes zero. However, for applications with non-trivial compulsory miss ratios, the MRC will never intersect the x-axis. Recall that a compulsory miss ratio is independent of the cache size. For such applications, the working set equals the cache size at which the MRC flattens out and becomes parallel to the x-axis.

4. The Pirate and the Bandit

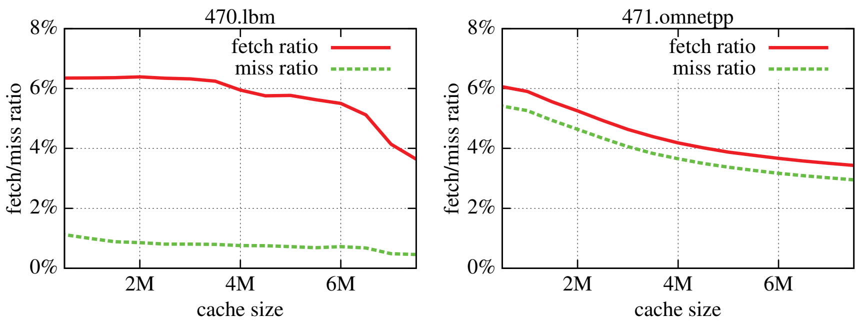
The main purpose of data locality analysis is to find opportunities to improve the miss ratio and ultimately the performance of the analyzed application. However, reducing the miss ratio of an application does not always improve its performance. High-end processors often implement latency-hiding techniques such as prefetching and out-of-order execution. While these techniques do not reduce the amount of data that has to be fetched from main memory, they do hide the latency (i.e., the service time) of the memory requests serviced by the higher levels of the memory hierarchy, thereby reducing the cost of cache misses. When increasing the cache capacity available to an application with a large degree of data locality, some of its memory accesses that originally resulted in cache misses are turned into cache hits. For example, this is the case for OMNet++. (See Figure 3.5). In terms of miss ratio, the application therefore utilizes the additional cache capacity. However, the latency of these memory accesses might already have been hidden by latency hiding techniques. Consequently, the application's performance might not significantly improve, despite the increased cache capacity and reduced miss ratio. From a performance point of view, it therefore does not utilize the additional cache capacity.

While MRCs provide the data needed to analyze an application's cache utilization in terms of miss ratio, they are not well suited for analyzing cache utilization in terms of performance. For this we need to know the application's performance (e.g., Cycles Per Instructions, CPI) as a function its available cache capacity. Furthermore, in order to analyzing an application's off-chip memory bandwidth utilization, we need to know its CPI as a function of its available off-chip memory bandwidth. This data allow us to determine how applications' performance are affected when increasing both their available cache capacity and off-chip memory bandwidth, and therefore to analyze their utilization of those two resources. To this end, this thesis makes two key contribution: Cache Pirating (Paper III) and Bandwidth Bandit (Paper IV). These two profiling methods enable us to measure any performance metric available through hardware performance counters as a function of the available cache capacity and off-chip memory bandwidth.

There are a number of options how to obtain data for analyzing applications' utilization of the memory hierarchy. Cycle-accurate simulators (such as [3, 7]) can provide data with high level of detail and can be extended to measure almost any imaginable performance metric. While simulations are extremely valuable for computer architecture design space explorations, it is not necessarily the best option for application performance analysis. In this case, the goal is often to analyze the application's performance on the actual target architecture. This would in many cases require the simulator to be validated against the target architecture. However, as this is not always required for design space exploration, few simulators are validated against real architectures. However, there are exceptions (e.g., [8]). Furthermore, software performance analysis should ideally be done with realistic working sets. This can inhibit the use of simulators as the simulation time might be unacceptable.

In the context of software performance analysis, an alternative to simulators are hardware performance counters. They are mainly designed to count a predefined set of performance related events, such as cache, TLB and branch predictor misses, which occur during the execution of an application. As the application is run on the real target hardware it does not suffer any slowdown. This enables the measurement of performance metrics while the applications run with large, realistic input sets, which is important for application performance analysis. Furthermore the measurements reflect the performance of the application on real hardware. However, as we pointed out in Chapter 2, the performance counters can only be used to measure cache misses and performance for the cache sizes of the caches in the memory hierarchy of the target computer. This makes it hard to determine whether an application benefits from increasing the cache capacity available to it, i.e. whether it can utilize any additional cache capacity.

The Cache Pirate (Paper III) and Bandwidth Bandit (Paper IV) enable the measurement of any performance metric available through performance counters as a function of the cache capacity and off-chip memory bandwidth available to the profiled application. This information allows us to determine how well an application utilizes both cache capacity and off-chip memory bandwidth, which are two important resources in the cache hierarchy. Both methods work by co-running a stress application with the target application (the one which is measured). These stress applications "steal" a configurable amount cache capacity, in the case of the Cache Pirate, and off-chip memory bandwidth, in the case of the Bandwidth Bandit, from the target application. As a result the target application only gets to use the remaining cache capacity and memory bandwidth. Measuring the target application while it is co-running with the stress applications enables us to measure any performance metric available through hardware performance counters, such as execution rate (IPC) and off-chip memory bandwidth (GB/s), as a function of both the cache capacity and off-chip memory bandwidth available to the target application.



(a) (b)

Figure 4.1. Fetch and miss ratio curves for 470.lbm (a) and 471.omnetpp (b).

4.1 Cache Pirate

Figure 4.1 shows the miss ratio and fetch ratio curves obtained using Cache Pirating for LBM and OMNet++ on a quad-core Intel Nehalem machine with an 8MB shared last-level cache. In these graphs, miss ratio is the ratio of memory accesses that result in cache block being fetched from main memory. These fetches are called demand fetches. The fetch ratio includes both demand fetches and (hardware and software) prefetches. It therefore accounts for all cache blocks fetched from main memory. As such, the fetch ratio is always greater than or equal to the miss ratio. In Chapter 3 we did not distinguish between the two. When prefetching is disabled, the miss ratio and fetch ratio are the same. However, when enabling prefetching, the fetch ratio might increase. This is because the prefetchers might mistakenly fetch cache blocks that are never used. Statstack does not model hardware prefetching. However, under the assumption that the fetch ratio is the same, regardless if prefetching is enabled or disabled, Statstack always reports fetch ratios. As this is often the case, the data provided by Statstack approximates the profiled application's fetch ratio.

Figure 4.1(a) shows the fetch ratio and miss ratio of LBM. There are several things to note in this graph. First, the miss ratio is much lower than the fetch ratio, which indicates that the hardware prefetchers are successfully prefetching useful data. This is expected as LBM performs a stencil computation with a very regular, and therefore easily prefetchable access, pattern. Second, despite the successful prefetching, LBM's miss ratio still increases when its available cache capacity is reduced. We would therefore expect LBM's CPI to increase accordingly. However, Figure 4.2(a), which reports LBM's CPI as a function of its available cache capacity, shows that this is not the case. Instead, LBM's CPI is independent on its available cache capacity. This is because LBM issues most of its loads in the beginning of each step (iteration) of the stencil computation. Furthermore, these loads are independent, and most

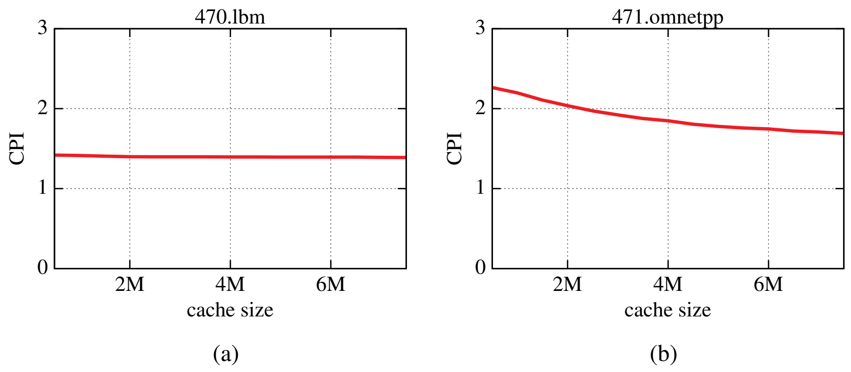


Figure 4.2. CPI as a function of cache size for 470.lbm (a) and 471.omnetpp (b).

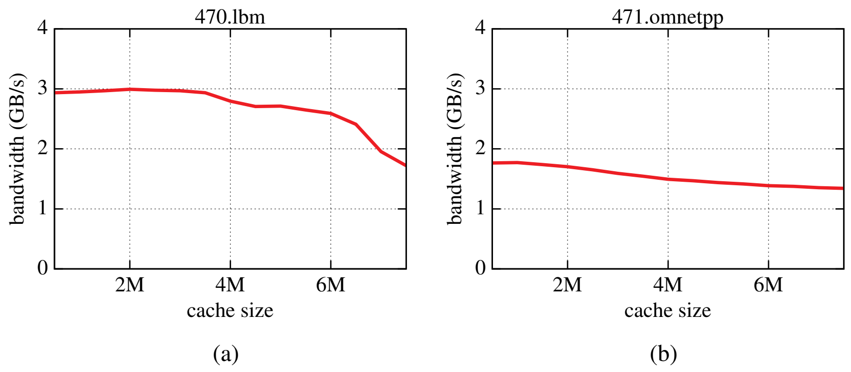


Figure 4.3. Bandwidth as a function of cache size for 470.lbm (a) and 471.omnetpp (b).

of the additional cache misses can therefore be issued in parallel. As a result, their latencies can be effectively hidden by out-of-order execution.

In terms of fetch ratio, LBM clearly utilizes additional cache capacity. Its fetch ratio improves when it receives more cache capacity. However, in terms of performance, LBM does not utilize the cache. This is shown in Figure 4.2(a), where LBM's CPI is virtually flat, indicating LBM is able to maintain a constant CPI when its available cache capacity is reduced. This, however, comes at the cost of an increased demand for off-chip memory bandwidth. Figure 4.3(a) shows LBM's off-chip memory bandwidth demand as a function of its available cache capacity. As LBM's fetch ratio increases when its cache capacity is reduced, the memory system has to supply it with data at a higher rate in order to maintain a constant execution rate. In the case of LBM, we can essentially trade cache capacity for off-chip memory bandwidth. In this context, Figure 4.3(a) can be viewed as reporting the exchange rate between cache capacity and off-chip memory bandwidth.

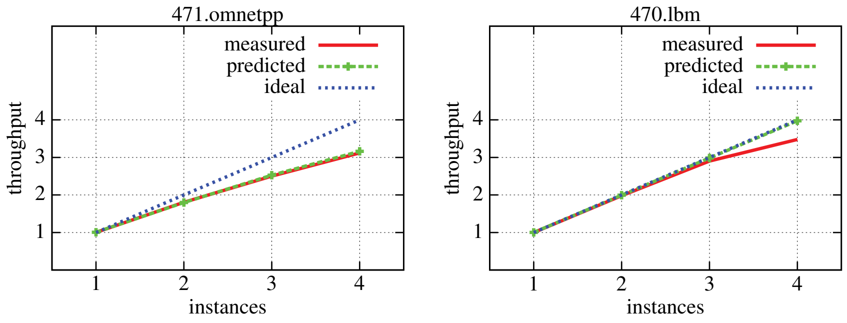


Figure 4.4. Aggregate throughput of multiple co-running instances of 471.omnetpp (a) and 470.lbm (b).

Figure 4.1(b) shows the fetch ratio and miss ratio of OMNet++, which, except for a small offset, are almost identical. This indicates that the hardware prefetchers are not able to fetch much of OMNet's data. This suggests the OMNet has a much less regular access pattern, than LBM. Consequently, its CPI, shown in Figure 4.2(b) does increase when its available cache capacity is reduced, and it OMNet++ therefore utilizes additional cache capacity, both in terms of fetch ratio and performance. Furthermore, as both its fetch ratio and CPI increase in concert when its available cache capacity is reduced, its off-chip memory bandwidth demand does not significantly increase. The increased demand for data fetches is offset by the reduced execution rate.

4.1.1 Case Study

To give a concrete example of how one can use the data obtained using Cache Pirating, this section presents a case study in which we analyze how the aggregate throughput scales when co-running multiple instances of LBM and OMNet++.

Figure 4.4(a) shows the aggregate throughput when running up to four instances of OMNetT++ on a quad core Intel Nehalem processor with an 8MB shared last-level cache. As the figure shows, the throughput does not scale perfectly. For example, when running four instances, we observed a total throughput of only three times that of a single instance. When running multiple instances of the same application, all instances typically receive equal portions of the shared cache capacity. When increasing the number of co-runners, the cache capacity available to each individual instance is therefore reduced.

In light of the analysis of OMNeT++ in the previous section, it is not surprising that its throughput does not scale perfectly. Since OMNeT++'s performance decreases when its available cache capacity is reduced, it does indeed utilize its cache capacity. When running four instances of OMNet++, each

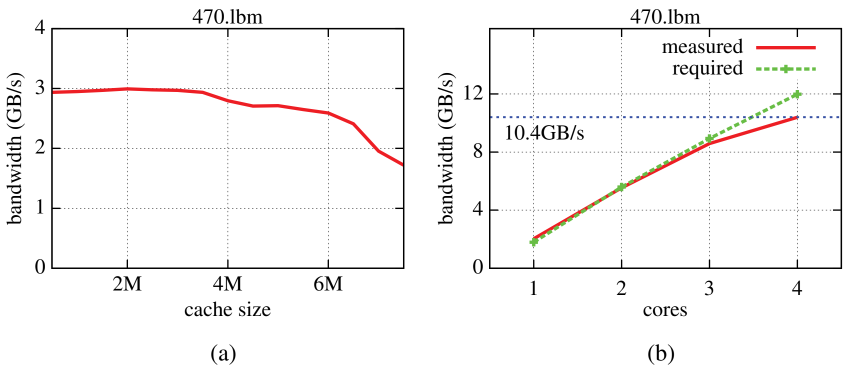


Figure 4.5. (a) Bandwidth as a function of cache size and (b) bandwidth demand for one, two, three and four co-running instances of 470.lbm.

instance will receive approximately 2MB of the shared cache capacity, and therefore run at a CPI of 2.0, or 20% slower than when they get to use all of the shared cache capacity. (See Figure 4.2(b).) Based on the CPI data in Figure 4.2(b) we can accurately predict the observed throughput scaling. This prediction is shown in Figure 4.4(a) (the graph labeled "predicted"). As the throughput prediction matches the measured throughput almost perfectly, we can conclude that the limited throughput is due to OMNeT's fairly good cache utilization.

LBM is a different case. In the previous section we saw that LBM does not utilize the cache at all, and we therefore expect its throughput to scale perfectly. However, as shown in Figure 4.4(b), this is not the case. To understand this, we look at LBM's off-chip memory bandwidth curve in Figure 4.5(a). (Figure 4.5(a) shows the same data as Figure 4.3(a) but is repeated here for convenience.) This data shows the off-chip bandwidth required by LBM to achieve the CPI in Figure 4.2(a). It shows that when LBM's available shared cache capacity is reduced it requires more bandwidth. Figure 4.5(b) shows the required and measured off-chip bandwidth for running up to four instances of LBM. As Figure 4.5(b) shows, when running three instances, each instance require 3GB/s, for a total of 9GB/s, which is less than the system's maximum bandwidth of 10.4GB/s. However, the required bandwidth to run four instances at the CPI shown in Figure 4.5(b) is 12GB/s, which is more than the system's maximum.

4.2 Bandwidth Bandit

In the previous section we saw that when co-running four instances of LBM, the memory system could not deliver the bandwidth required to achieve perfect

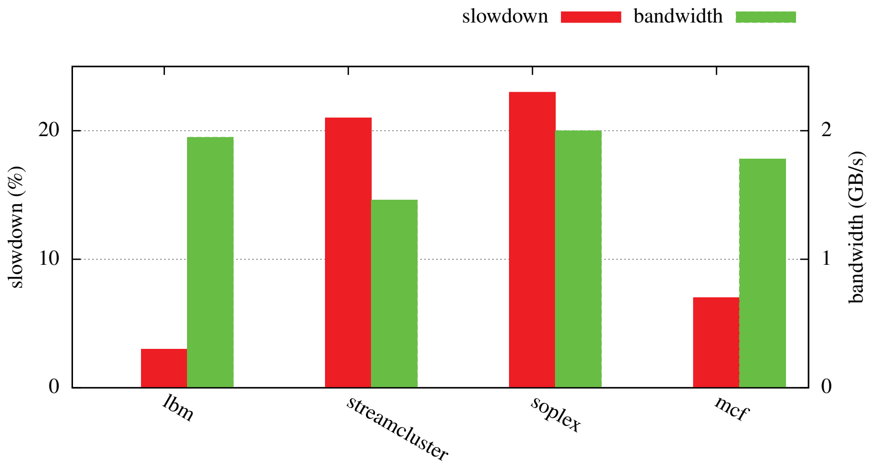


Figure 4.6. Slowdown due to memory contention (left vertical axis) and bandwidth demand (right vertical axis) of four single-threaded applications. The slowdown is the execution when the applications experience memory contention relative to that when they are running alone (i.e., they have exclusive access to all off-chip memory resources).

(linear) throughput scaling. This bandwidth limitation is due to contention for off-chip memory resources, which is the topic of this section.

When co-running multiple application/threads, they contend for the shared memory resources, e.g., memory controllers; data and address buses; and DRAM banks. In terms of performance, this has two main effects. First, it limits the bandwidth available to the instances. This is largely due to limits on the number parallel memory requests that can be kept in-flight at various levels in the memory hierarchy. Second, it increases accesses latencies. This is due to several factors, for instance: DRAM bank conflicts, which occur when two or more instances are trying to access the same bank at the same time; and page-cache conflicts, which occur when a row cached on behalf of one application/thread is evicted in order to cache a row of another application/thread.

As mentioned above, contention for off-chip memory resources affect both the bandwidth available to and latencies achieved by the co-running application. To further complicate matters, different applications can have significantly different sensitivities to reduced bandwidths and increased latencies. Figure 4.6 show the slowdown of four applications when experiencing contention for off-chip memory resources. The four applications are co-run with the same set of co-runners, and therefore experience the same amount of memory contention. However, Streamcluster and Soplex have much larger slowdown than LBM and MCF. The figure also shows the applications' bandwidth demands. It would seem reasonable that applications with higher bandwidth demands are more sensitive to contention as they consume more memory resources. However, Figure 4.6 clearly shows that this is not the case. In the

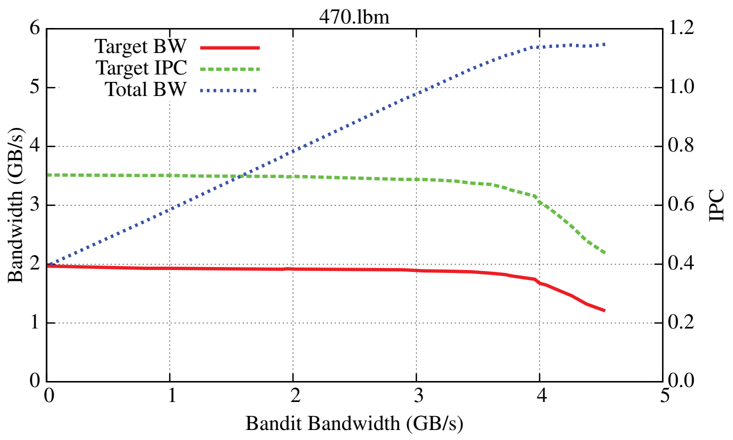


Figure 4.7. Data obtained using the Bandwidth Bandit.

context of software performance analysis, it is therefore important to quantify applications' sensitivities to contention for off-chip memory resources.

Cache Pirating enables us to measure the profiled application's demand for off-chip memory bandwidth. However, as shown in Figure 4.6, this data does not indicate how the application's performance is impacted by reduced bandwidths and increased latencies. To analyze this we therefore need different data. However, as mentioned above, there are many shared resources in the off-chip part of the memory hierarchy. Analyzing the contention for each of these resources individually, and then inferring their overall performance impact, can be intractable. Ideally, we want to summarize their total performance impact with a single metric. The amount of memory contention generated by an application is generally higher for applications that generate more off-chip memory traffic, i.e., applications that consume more off-chip memory bandwidth. In this section, we therefore study how an application's performance is impacted as a function of how much off-chip memory bandwidth their co-runners consume.

Figure 4.7 shows the data obtained using the Bandwidth Bandit for LBM. To get this data we co-run LBM with the stress application (the Bandit) that consumes a configurable amount of off-chip memory bandwidth and measured the bandwidth and IPC achieved by LBM. To highlight the impact of bandwidth limitations, this data was obtained on a machine with only one out of three memory channels activated. This roughly reduces the available bandwidth with a factor of three. As we want to measure the impact due to contention for off-chip memory resources in isolation, we ensure the Bandit application accesses memory in way that such that it consumes a trivial amount of cache capacity (for further details see Paper IV). Figure 4.7 has three curves; Target BW, the target application's, in this case LBM's, bandwidth; Target IPC, the target application's IPC; and Total BW, the sum of the Target's and the Bandit's bandwidths. The scales for the two bandwidth curves (Target BW and Total BW) are shown on the left vertical axis, while the Target IPC is shown on the right vertical axis. The horizontal axis shows the Bandit's bandwidth.

There are several things to note in Figure 4.7. First, the total bandwidth (labeled "Total BW") levels out at about 5.7GB/s. This is the saturation bandwidth (i.e., maximum achievable aggregate bandwidth) for this set of co-runners (LBM and Bandit). This occurs when the Bandit's bandwidth reaches about 3.8GB/s. Further increasing the Bandit's bandwidth reduces the bandwidth achieved by LBM so that the total bandwidth remains at 5.7GB/s. Second, when LBM runs alone (i.e., not co-running with the Bandit) its bandwidth is about 2GB/s. When the bandwidth consumed by the Bandit is increased (moving to the right along the horizontal axis), LBM's bandwidth stays at this level (2GB/s) until the Bandit's bandwidth reaches the point at which the total bandwidth levels out. Third, LBM's IPC curve follows the same trend as its bandwidth curve. As the Target's fetch ratio is not impacted by the Bandit, the Target's bandwidth is determined by its IPC ($bandwidth = IPC \times fetch\_ratio$). Finally, the saturation bandwidth depends on the set of co-runners. However, we can always find it by identifying the level at which the total bandwidth levels out.

Based on the above observations we can draw the following conclusions. First, even at fairly modest bandwidths, the contention generated by the Bandit causes the Target's access latencies to increase. However, LBM's performance is not negatively affected before the off-chip memory bandwidth saturates. This indicates that LBM's performance is not impacted by increased access latencies. Second, once the bandwidth saturates, LBM's performance drops proportionally to the bandwidth consumed by its co-runners. This allows us to explain why LBM does did not achieve perfect (linear) scaling in the case study presented in Section 4.1.1. Based on Figure 4.5(b) we know that the total bandwidth demand of four co-running instances of LBM is 12GB/s. However, the total bandwidth actually measured when running four instances is 10.4GB/s (see Figure 4.5(b)). This suggests that the four instances saturate the off-chip memory bandwidth. Now, since the bandwidth is saturated and the achieved bandwidth is 87% of the demand, we expect the each LBM instance to run 13% slower than when the bandwidth is not saturated. The IPC of LBM before saturation is about 1.4 (see Figure 4.2(a)) the aggregate IPC of four instances is therefore 4.9 ($4 \times 0.87 \times 1.4$), and the resulting normalized throughput is 3.5 ($4.9/1.4$). This is very close match to the measured throughput and therefore shows that we can indeed use what we learned about LBM from analyzing its bandwidth data to predict this sub-linear scaling.

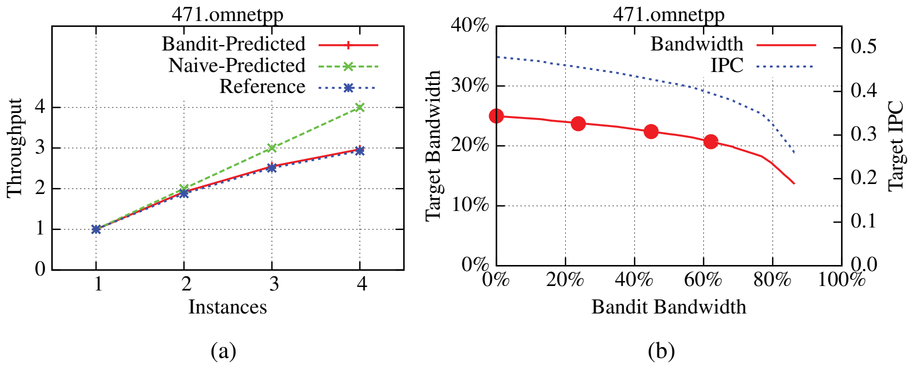


Figure 4.8. Relative throughput (left) and Bandwidth Bandit data of OMNet++.

4.2.1 Case Study

In this section we present a case study in which we show how the data obtained using the Bandwidth Bandit can be used to estimate the throughput when co-running multiple instances of OMNet++.

In order to factor out the impact of cache sharing, we partitioned the 8MB shared, last-level cache on our quad-core Nehalem machine so that each co-running instance always gets 2MB of cache capacity each. To achieved this we patched the Linux kernel to add support for page coloring. This allows us to study the impact of contention for off-chip memory resources in isolation.

Figure 4.8(a) show the normalized aggregate throughput of OMNet++. It clearly does not scales perfectly (i.e., linearly). To investigate this we look at OMNet's bandwidth graphs in Figure 4.8(b). In this figure, the bandwidths are expressed as a percentage of the saturation bandwidth. This data shows that, contrary to that of LBM, the IPC of OMNet++ does increase long before the bandwidth saturates, which indicates that it OMNet++ is sensitive to increased access latencies.

Given the bandwidth data of an application, finding the throughput of a given number of co-running instances is a two-step process. First, we use the bandwidth graphs to find the bandwidths of the co-running instances. Then, we use these bandwidths to find the co-running instances' individual IPCs, which give us the finally overall throughput.

Bandwidth: When co-running multiple instances of the same application, all instances get an equal amount of bandwidth, finding the bandwidth of one instance therefore gives us the bandwidth of all instances. Finding how much bandwidth one of the co-running instances get amounts to finding the (x, y)-point on its bandwidth graph where $y = x$, $y = 2x$ and $y = 3x$ for two, three and four co-running instances, respectively. For example, the y value which is the solution to $y = 2x$ is the bandwidth of one instance, when its co-running with two instance that together consumes twice the bandwidth of the first instance.

The above equations can be easily solved using standard fixed-point methods. Their solutions of are marked with circles in Figure 4.8(b).

Throughput: Once we know the bandwidth of all instances, we can simply read out their IPC in Figure 4.8(b), and the aggregate through put is then the sum of the instances IPCs.

Figure 4.8(a) show the throughput predicted using the Bandwidth Bandit ("Bandit-Predicted") and the measured reference throughput ("Reference"). It also show a "naive" throughput prediction ("Naive-Predicted"). Because the bandwidth demand of OMNet++ is slightly less than 25% of the saturation bandwidth, which is less than the system peak bandwidth, we would not expect four instances to saturate the bandwidth. (Their total bandwidth demand is less than 100% of the saturation bandwidth.) Therefore, without any additional information, our best (although naive) prediction would have been that the throughput scales linearly.

5. Speedup Stacks

A key property of multi-threaded programs is how their execution times scale when increasing the number of threads. This can be illustrated in a speedup graph which reports a program's speedup as a function of its number of threads. See Figure 5.1. For multi-threaded programs to fully utilize the increasing core counts of present and future multi-core processors, their performance has to scale linearly with the number of threads. However, there are several factors that can limit the scalability of multi-threaded programs. Besides Amdahl's law, Eyerman et al. [12] identified the following main scalability bottlenecks: contention for cache capacity and off-chip memory resources; cache coherency; synchronization; load imbalance and parallelization overheads. The program shown in Figure 5.1 does not achieve perfect (linear) speedup. For example, it only achieves a speedup of 2.5 with four threads.

While a speedup graph shows how a multi-threaded program scales, it does not alone provide enough information to determine why the program scales in this way. To this end Eyerman et al. [12] introduced the concept of the speedup stack, illustrated in Figure 5.1. The speedup stack reports how much each of the scalability bottlenecks (listed above) limits the program's speedup. It consists of several components, one for each scalability bottleneck. The heights of the components represent how much the program's speedup is reduced due to the corresponding scalability bottlenecks. The scalability of the program in Figure 5.1 is limited by three bottlenecks: contention for cache capacity (cache), contention for off-chip memory (memory) and synchronization (synch). For this program, the largest component is cache, and the major factor limiting its speedup is therefore contention for cache capacity. This suggests that the most effective way to increase its scalability is to reduce the contention for cache capacity. A software developer could, for example, improve data locality, while a hardware architect could increase cache sizes.

Heirman et al. [15] and Eyerman et al. [12] both present methods for obtaining speedup stacks. Heirman's method requires simulations, while Eyerman's rely on custom hardware performance counter support. While they report encouraging results, both methods have drawbacks which we address in this paper. First, accurately simulating real multi-core hardware is a big challenge. For example, besides being relatively slow, it might require detailed and undisclosed information about the simulated multi-core. Second, as of yet none of the major chip manufacturers implement the performance counting hardware required by Eyerman's method.

In Paper V we present a software-only profiling method for obtaining speedup stacks on commodity multi-cores. The method targets symmetric, fork-join

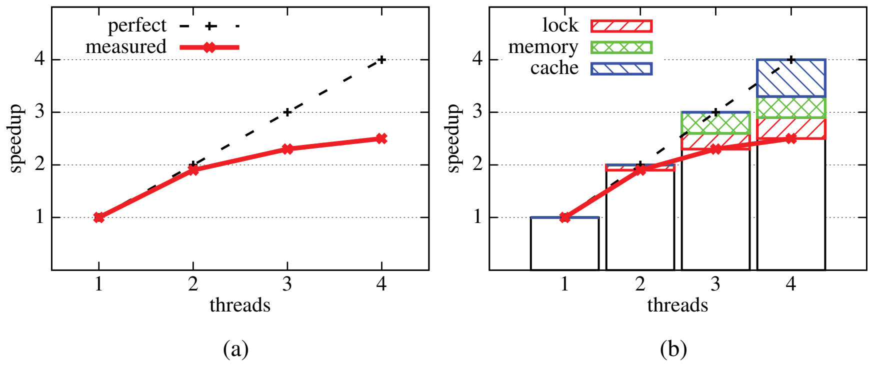


Figure 5.1. Speedup graph (a) and corresponding speedup stack (b).

style parallel programs with limited data sharing. This includes many data parallel applications. The method captures speedup stacks for each parallel phase (i.e., between a fork and the corresponding join) of the profiled application. The resulting speedup stacks report three components: 1) contention for cache capacity; 2) contention for off-chip memory resources; and 3) synchronization, which includes lock contention, barrier synchronization and load imbalance. These components represent the main scalability bottlenecks of the targeted class of programs.

5.1 Speedup Stacks

We define the speedup stack in terms of the speedup that a multi-threaded program would achieve when run on a hypothetical machine where the scalability bottlenecks can be eliminated one-by-one, but which is otherwise identical to a real, commodity multi-core. This hypothetical machine can be thought of as follows. To eliminate the bottlenecks due to synchronization, all threads should perform as if they never had to wait for the other threads. For example, in the case of lock contention, each thread should perform as if the other threads never held the lock in question. To eliminate the scalability bottlenecks due to contention for shared resources (cache capacity and off-chip memory resources), all threads should perform as if they had exclusive access to the shared resource.

To this end we develop a framework that allow us to profile each individual thread of a mulit-threaded program while running them in an environment where we can control the presence/absence of the scalability bottlenecks. The basic idea is to first eliminate all scalability bottlenecks and reintroduce them one-by-one. In order to eliminate the scalability bottlenecks, we run the profiled thread alone and thereby giving it access to all shared resources, e.g., cache capacity, off-chip memory resources and critical sections. This is

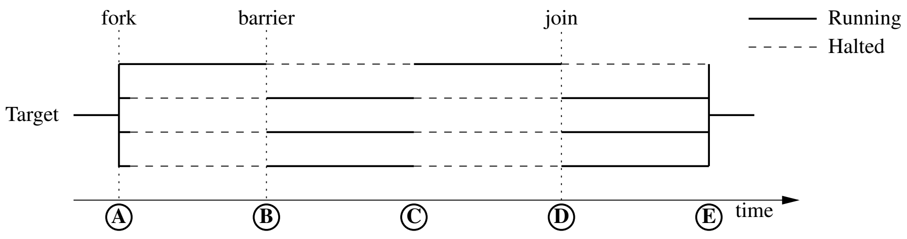


Figure 5.2. Chart showing how the profiling framework works. Ⓐ Halt all but the profiled thread. Ⓑ The profiled thread reached a barrier, start the other threads and let them catch up. Ⓒ At this point, all threads have reached the barrier. Repeat A until the join is reached. Ⓓ The profiled thread reached the join, halt it and let the other threads catch up. Ⓔ All threads have made it through the parallel phase. Profiling done.

achieved as follows. At the beginning of a parallel phase we halt all but one thread which we profile, and let it run alone until it reaches a synchronization barrier. At this point, we halt the profiled thread and let the others catch up. When all threads have reached the synchronization barrier, we again halt all but the profiled thread and repeat the above process. Alternating the two sets of threads between barriers guarantees a correct execution in terms of synchronization. This is illustrated in Figure 5.2. Furthermore, as the profiled thread runs alone, it gets exclusive access to the shared cache capacity and the off-chip memory resources; and does not experience any lock contention, the other threads do not hold any locks while waiting at the barrier.

To introduce scalability bottlenecks in the above environment, we use techniques such as Cache Pirating (Paper III). Cache Pirating lets us limit the cache capacity available to the profiled thread. For example, this allows us to emulate an environment in which the profiled thread has access to a limited amount of shared cache capacity, but at the same time has access to all off-chip memory resources and does not have to wait for access to locks. For further details please see Paper V.

5.2 Case Study 1: Single-threaded applications

In this section we present speedup stacks for multiple co-running instances of single threaded applications. Figure 5.3 shows the speedup stacks of OMNet++ and LBM. These speedup stacks were obtained on a quad-core Intel Nehalem machine. As each instance performs the same amount of work no matter how many are co-running, we look at aggregate throughput rather than speedup. Furthermore, as they have separate address spaces and do not synchronize, the only two scalability bottlenecks that can limit their throughput are contention for cache capacity and off-chip memory resources, and their speedup stacks will therefore only consist of the corresponding two compo-

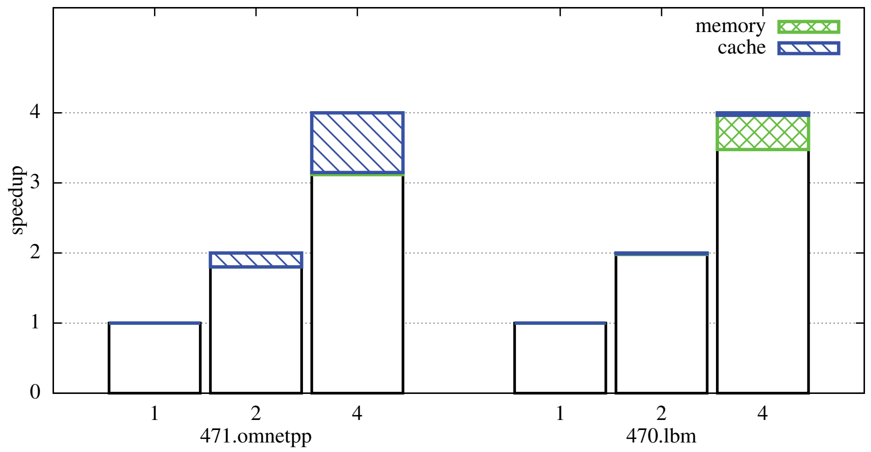


Figure 5.3. Speedup stack of co-running instances of OMnet++ (a) and LBM (b).

nents - cache and memory. In the following we briefly discuss how to interpret OMNet++'s and LBM's speedup stacks and how their throughputs can be improved.

Figure 5.3 shows OMNet's speedup stack. It reveals that OMNet's throughput is limited mainly by contention for cache capacity. This is consistent with the results of the analysis presented in Section 4.1.1. There are two options to improve OMNet's throughput, either improve data locality and thereby reduce its working set; or increase the cache size. To achieve perfect scaling, the CPI of each co-running instance has to be the same as that of a single instance running alone, which is about 1.70 (see Figure 4.2(b)). Consider the case of four co-running instances. If each instance gets 6MB of cache, their CPIs would be about 1.73. This corresponds to a relative aggregate throughput of 3.93 ($1.70/1.73 \times 4$). Therefore, reducing the working set size by approximately 4MB or increasing the cache capacity available to each instance by 4MB (i.e., increasing the cache size by 16MB ($6 \times 4 - 8$MB)), would result in close to perfect scaling up to four co-running instances.

Figure 5.3 shows LBM's speedup stack. LBM's throughput is limited by contention for off-chip memory resources. In Section 4.1.1 we drew the following conclusions about LBM. First, as its CPI is virtually flat, LBM is not directly affected by how much cache capacity it receives, and will scale perfectly as long as the off-chip memory bandwidth demands of the co-running instances are met. LBM's bandwidth demand when co-running four instances is 12GB/s (4×3GB/s). However, when co-running four instances, the aggregate bandwidth actually achieved is only 10.6GB/s, which is less than the demand (see Figure 4.5(b)). This behaviour is reflected in LBM's speedup stack.

To improve LBM's throughput there are two options, either increase the available off-chip memory bandwidth or reduce the bandwidth demand. First,

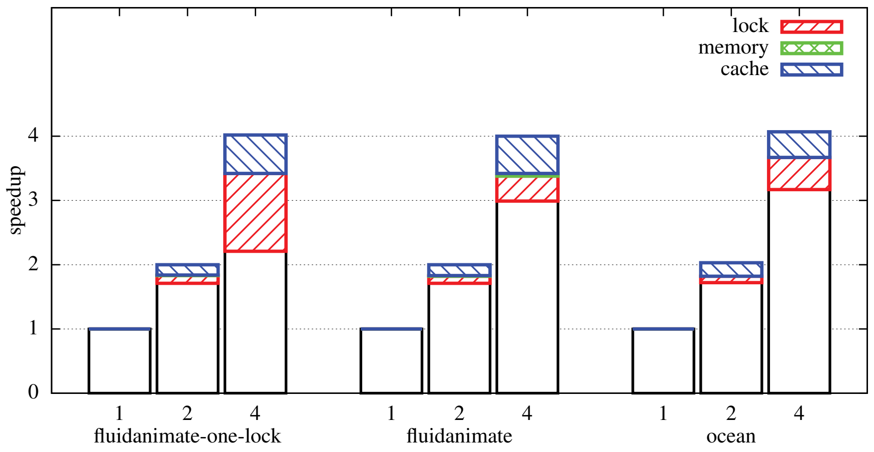


Figure 5.4. Speedup stacks for three data parallel applications.

as the bandwidth demand is 12GB/s, increasing the available off-chip memory resources such that a bandwidth of 12GB/s can be achieved would result in perfect (linear) scaling. Second, Figure 4.3(a) shows that increasing the cache size by 10MB (4×2.5MB) would reduce the bandwidth demand of each instance to about 2.7GB/s. This would result in an aggregate demand of 10.8GB/s (4×2.7GB/s) which is just about what three memory channels can sustain. Increasing the cache size by 10MB would therefore result in perfect (linear) throughput scaling.

5.3 Case Study 2: Multi-threaded applications

Figure 5.4 shows the speedup stacks, including synchronization components, for the three data parallel benchmarks from. These three benchmarks are the ones with the largest synchronization overheads that we could find that do not use condition variables (which we currently do not support)[1]. We present data for two versions of fluidanimate. Fluidanimate iteratively computes over a set of particles in a three dimensional space which is split into sub-spaces, one for each thread. In a single iteration several different properties are computed for each particle, between which the threads synchronize using a single barrier. In the vanilla version of fluidanimate, each particle has its own lock. When a thread computes on a particle residing on a boundary of its sub-space, it has to grab the lock associated with the particle's neighbors. To introduce more lock contention we created a version of fluidanimate that uses a single lock (fluidanimate-one-lock).

[1]We measured the synchronization overheads of all data parallel benchmarks in PARSEC [6], NAS [23] and SPLASH2 [32] that do not use condition variables. With four threads only fluidanimate and ocean had significant significant synchronization overheads.

There are several things to note. Figure 5.4 shows that both fluidanimate versions have large lock components, and that fluidanimate-one-lock version lock component is larger than vanilla fluidanimate, as expected. The cache component is slightly smaller for the vanilla version. As there is more lock contention in the single lock version, the threads spend more time spinning waiting for access to the lock. During this time they do not utilize the shared cache capacity, which allows the non-spinning threads access to more cache capacity. This causes fluidanimate-one-lock to have a slightly smaller cache component than vanilla fluidanimate.

Finally, we note that all three benchmarks in Figure 5.4 have virtually zero memory components. This expected as their memory bandwidth demands at four threads are low (1.8GB/s, 1.9GB/s and 1.7GB/s, respectively) and can be easily satisfied. For our method to predict a memory component of zero, the speedups estimated using Cache Pirating and the method to estimate synchronization overheads, presented above, must match up. This is indeed the case, and indicates that both these methods make accurate predictions, as it is unlikely that both methods have errors that balance each other to produce a zero memory component.

6. Summary and Conclusions

The performance of many applications is largely determined by how well they utilize the resources in the memory hierarchy. For such applications, there are two principal approaches to improve performance: optimize the memory hierarchy and optimize the software. In both cases, it is important to both qualitatively and quantitatively understand how the software interacts with the memory hierarchy. To this end, this thesis presents several novel profiling methods for memory-centric software performance analysis. The goal of these profiling methods is to provide general, high-level, quantitative information describing how the profiled applications utilize the resources in the memory hierarchy. For such techniques to be broadly applicable the data collection should have minimal impact on the profiled applications while at the same time not be dependent on custom hardware and/or operating system extensions.

In this thesis we mainly focus on the following two key resources: caches and off-chip memory bandwidth. A general way to present how well an application utilizes these resources is to plot performance metrics, such as miss ratio, fetch ratio and CPI, as a function of how much of the resources the application receives. This thesis shows that such high-level profiling information can indeed be accurately obtained with little impact on the profiled application and without requiring costly simulations or custom hardware support. To obtain this type of information, we propose and evaluate three approaches: Statstack, the Cache Pirate and the Bandwidth Bandit. Statstack, which is mainly geared towards data locality analysis, employs statistical sampling techniques and estimates the profiled application's miss ratio curve. The profiling data required by Statstack can be obtained at low cost by using hardware performance counters and watchpoints. The latter two methods, Cache Pirate and the Bandwidth Bandit, use hardware performance counters to directly measure the profiled application while it is running on a commodity multi-core. As a result, they capture the profiled application's behavior on real, commodity hardware with a limited impact on the profiled applications' execution times.

While the main focus of this thesis is on the design and evaluation of the core profiling and how to interpret the resulting data in a general context, we presents several case studies. In these case studies, we show how to apply the obtained profiling data to analyze both the throughput when co-running multiple instances of single-threaded applications and the scalability of muti-threaded applications on a commodity multi-core. Furthermore, the type of data captured by the profiling methods presented in this thesis have been successfully used by Acumem ThreadSpotter [2].

7. Introduction in Swedish

Utvecklingen inom halvledarindustrin har under de senaste decennierna gått otroligt fort men tyvärr har inte all typer av datorkomponenter utvecklats i samma hastighet. Till exempel har processorkärnornas klockfrekvens generellt utvecklats mycket snabbare än primärminnenas frekvens, vilket kan leda till problem när dessa ska användas tillsammans. När processorkärnornas hastighet ökar, ökar normalt sett även hastigheten med vilken de behöver läsa och skriva data till primärminnet. Om primärminnet inte klarar av att leverera data i den takt som processorkärnorna kräver, kommer hela datorsystemets hastighet att begränsas av primärminnets hastighet.

På senare tid har processorernas frekvensutveckling börjat plana ut. Detta beror huvudsakligen på att deras effektförbrukning har nått ohållbara nivåer. För att dra nytta av den fortsatta ökningen av antalet transistorer på halvledarchipen har datorindustrin istället gått över till att integrera flera processorer på samma chip. Idén bakom detta är att två processorer på en given tid kan utföra samma mängd jobb som en enda processor med dubbla frekvensen. I teorin skulle därför beräkningstiden kunna halveras genom att dubblera antalet processorer. Förhoppningen är att kunna öka antalet processorer i en sådan takt att kraven på ökad beräkningshastighet kan mötas utan att nämnvärt öka processorernas frekvens, vilket skulle leda till att effektförbrukningen kunde hållas nere. Men, eftersom två processorer efterfrågar dubbelt så mycket data som en processor, givet att deras frekvenser ar desamma, blir resultatet fortfarande att datorsystemens hastighet kan begränsas av primärminnets hastighet.

Till vilken grad datorsystemets hastighet är begränsad av primärminnet beror på de individuella datorprogrammens krav på primärminnet. Många av dagens datorprogram läser och skriver stora mängder data. Exekveringstiden för många av dessa program är därför ofta till en hög grad beroende av hur snabbt minnessystemet kan leverera data. Det leder till att moderna datorsystem måste ha både stora och snabba primärminnen. Det finns många olika typer av minnen som kan användas i ett minnessystem, med olika hastighet och kostnad. En enkel lösning skulle vara att endast använda den snabbaste typen av minne men detta skulle kunna resultera i en onödigt hög tillverkningskostnad. Den traditionella lösningen är därför att bygga hierarkiska minnessystem.

Under en given tidsperiod läser och skriver datorprogram normalt sett endast en relativt liten delmängd av deras totala datamängd. Den här observationen kallas datalokalitetsprincipen. Hierarkiska minnessystem drar nytta av datalokalitetsprincipen genom att använda små, snabba minnen, så kallade cacheminnen, för att spara den mest frekvent använda datan och stora, långsamma minnen for att spara den mindre frekvent använda datan. På så vis kommer de flesta läsningar och skrivningar hanteras av de små, snabba cacheminnena vilket gör att minnessystemet blir signifikant snabbare, samtidigt som kostnaden hålls nere. Figur 2.1 visar en minneshierarki liknande den som finns i Intels Nehalem-processorer.

Graden av datalokalitet, dvs. storleken på den mest frekvent använda datamängden, varierar ofta mellan olika program. Olika program ställer alltså olika krav på minneshierarkin. I optimeringen av minneshierarkier drar man även nytta av egenskaper som programmens läs- och skrivmönster, vilka också kan variera mellan olika program. Både datorarkitekter, som designar minneshierarkier, och mjukvaruutvecklare, som skriver högprestandaprogram, kan därför dra nytta av kvalitativ och kvantitativ kännedom om programmens krav på minneshierarkin.

Den här avhandlingen presenterar flera nya profileringsmetoder för minneshierarkicentriska prestandautvärderingar. Målet med dessa profileringsmetoder är att generera generell, lättolkad, kvantitativ information som beskriver hur väl det profilerade programmet utnyttjar cacheminnena i minneshierarkin. Datorarkitekter kan använda sådan information for att avgöra hur många och hur stora cacheminnen minneshierarkin bör innehålla. Mjukvaruutvecklare kan använda sådan information for att avgöra om någon eller några av datastrukturerna inte får plats i cacheminnet och på så sätt identifiera de datastrukturer som, om möjligt, bör bytas ut eller på annat sätt ändras.

För att den här typen av profileringsmetoder ska vara användbara, krävs det att datainsamlingen har minimal påverkan på det profilerade programmet. Detta är viktigt av två huvudsakliga anledningar: För det första innebär minimal påverkan att tiden det tar att profilera programmet blir kort. Detta tillåter profilering av riktiga program som annars skulle kunna ta flera dagar att profilera. För det andra, i de fall då profileringen sker genom att mäta det profilerade programmets egenskaper medan det exekverar på riktig hårdvara innebär minimal påverkan att profileringsdatan representerar det profilerade programmets verkliga beteende. Om profileringsmetoden påverkar programmets beteende på ett oönskat sätt kan detta distordera den uppmätta profileringsdatan. För att vara generellt tillämpbara är det även önskvärt att den här typen av profileringsmetoder inte kräver hårdvarustöd som inte finns tillgängligt i dagens processorer.

Profileringsmetoderna som presenteras i den här avhandlingen kan ge en mängd olika prestandamått som funktioner av hur stor del av cacheminnena en applikation använder. Den här informationen beskriver bland annat de profilerade programmens marginella förbättring när cacheminnenas respektive storlekar ökas och beskriver på så vis hur väl den profilerade applikationen utnyttjar cacheminnena. Vidare har de presenterade metoderna minimal oönskad inverkan på de profilerade programmen, utan att kräva specialdesignad hårdvara. Flera fallstudier presenteras, där profileringsmetoderna används för att analysera prestandakonsekvenserna av att köra flera parallella instanser av enkeltrådade program på flerkärniga processorer med delade cacheminnen. Det visas även hur profileringsmetoderna kan kombineras för att tillåta analys av hur skalbarheten hos multitrådade program påverkas av resursbegränsningar i minneshierarkin.

8. Acknowledgement

This thesis is a result of collaboration. It had not been written without the support and encouragement from many. To begin I would like to thank my supervisors Erik Hagersten and David Black-Schaffer: Erik, for all the interesting and challenging discussion and encouragement. David, for encuraging me to trust my own instincts and for the countless hours he spent improving the presentation in our papers. Many thanks go to my co-author Nikos Nikoleris who provided great insights and spent many late nights working on the Cache Pirating, Bandwidth Bandit and Speedup Stack papers. Furthermore, I would like to thank the other members of the Uppsala Architecture Research Team: Andreas Sandberg, Muneeb Khan and Andreas Sembrant for all interesting discussions and for creating an inspiring work environment. I would also like to thank Mikael “Lax” Laaksoharju for proofreading the introduction in Swedish.

This work was supported in part by the Swedish Foundation for Strategic Research through the CoDeR-MP (Computationally Demanding Real-Time Applications on Multicore Platforms) project and by the Swedish Research Council within UPMARC (Uppsala Programming for Multicore Architecture Research Center).

References

[1] A. Fedorova, M. Seltzer and M. D. Smith. A Non-Work-Conserving Operating System Scheduler for SMT Processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), in conjunction with ISCA-33*, Boston, MA, USA, June 2006.

[2] Acumem. Acumem threadspotter$^{\text{TM}}$.
`http://www.roguewave.com/portals/0/products/threadspotter/docs/2011.1/manual/index.html`.

[3] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35, 2002.

[4] Erik Berg and Erik Hagersten. Statcache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the 2004 IEEE international symposium on performance analysis of systems and software*, ISPASS, Austin, Texas, USA, 2004.

[5] Erik Berg and Erik Hagersten. Fast data-locality profiling of native execution. In *Proceedings of ACM SIGMETRICS 2005*, Banff, Canada, 2005.

[6] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2), 2011.

[8] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, 2011.

[9] Peter J. Denning. The locality principle. *Commun. ACM*, 48(7), 2005.

[10] Peter J. Denning and Stuart C. Schwartz. Properties of the working set model. *Commun. ACM*, 15(3), 1972.

[11] Chen Ding and Yutao Zhong. Predicting Whole-Program Locality Through Reuse Distance Analysis. *SIGPLAN Not.*, 38(5), 2003.

[12] Stijn Eyerman, Kristof Du Bois, and Lieven Eeckhout. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *ISPASS*, 2012.

[13] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A Mechanistic Performance Model for Superscalar out-of-order Processors. *ACM Trans. Comput. Syst.*, 27(2):1–37, 2009.

[14] Erik Hagersten, Mats Nilsson, and Magnus Vesterlund. Improving cache utilization using acumem VPE. In *Parallel Tools Workshop*, 2008.

[15] W. Heirman, T. E. Carlson, Shuai Che, K. Skadron, and L. Eeckhout. Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In *IISWC*, 2011.

[16] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[17] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. Comput.*, 38(12), 1989.

[18] Tejas S. Karkhanis and James E. Smith. A First-Order Superscalar Processor Model. *SIGARCH Comput. Archit. News*, 32(2):338, 2004.

[19] Yul H. Kim, Mark D. Hill, and David A. Wood. Implementing Stack Simulation for Highly-Associative Memories. *SIGMETRICS Perform. Eval. Rev.*, 19(1), 1991.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI, Chicago, IL, USA, 2005.

[21] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *Computer*, 35, 2002.

[22] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2), 1970.

[23] NASA. NAS parallel benchmarks. `http://www.nas.nasa.gov/publications/npb.html`.

[24] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42, 2007.

[25] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: challenges in and avenues for CMP scaling. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA, 2009.

[26] Daniel Sanchez and Christos Kozyrakis. The ZCache: Decoupling ways and associativity. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2010.

[27] Andreas Sandberg, David Eklöv, and Erik Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, 2010.

[28] Andreas Sembrant, David Black-Schaffer, and Erik Hagersten. Phase guided profiling for fast cache modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO, San Jose, California, 2012.

[29] Andreas Sembrant, David Eklov, and Erik Hagersten. Efficient software-based online phase classification. In *Proceeding of the International Symposium on Workload Characterization*, IISWC, Austin, Texas, 2011.

[30] Rabin A. Sugumar and Santosh G. Abraham. Efficient Simulation of Caches Under Optimal Replacement with Applications to Miss Characterization. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, 1993.

[31] Tarek M. Taha and Scott Wills. An Instruction Throughput Model of Superscalar Processors. *IEEE Trans. Comput.*, 57(3):389–403, 2008.

[32] Ioannis E. Venetis. Modified SPLASH-2. `http://www.capsl.udel.edu/splash/`.

[33] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1), March 1995.

[34] Yutao Zhong, Steven G. Dropsho, Xipeng Shen, Ahren Studer, and Chen Ding. Miss Rate Prediction Across Program Inputs and Cache Configurations. *IEEE Trans. Comput.*, 56(3), 2007.