A Profiling Method for Analyzing Scalability Bottlenecks on Multicores

David Eklov, Nikos Nikoleris and Erik Hagersten
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
{david.eklov, nikos.nikoleris, eh}@it.uu.se

ABSTRACT

A key goodness metric of multi-threaded programs is how their execution times scale when increasing the number of threads. However, there are several bottlenecks that can limit the scalability of a multi-threaded program, e.g., contention for shared cache capacity and off-chip memory bandwidth; and synchronization overheads. In order to improve the scalability of a multi-threaded program, it is vital to be able to quantify how the program is impacted by these scalability bottlenecks.

We present a software-only profiling method for obtaining *speedup stacks*. A speedup stack reports how much each scalability bottleneck limits the scalability of a multi-threaded program. It thereby quantifies how much its scalability can be improved by eliminating a given bottleneck. A software developer can use this information to determine what optimizations are most likely to improve scalability, while a computer architect can use it to analyze the resource demands of emerging workloads.

The proposed method profiles the program on real commodity multi-cores (i.e., no simulations required) using existing performance counters. Consequently, the obtained speedup stacks accurately account for all idiosyncrasies of the machine on which the program is profiled. While the main contribution of this paper is the profiling method to obtain speedup stacks, we present several examples of how speedup stacks can be used to analyze the resource requirements of multi-threaded programs. Furthermore, we discuss how their scalability can be improved by both software developers and computer architects.

1. INTRODUCTION

The increasing core counts on virtually all types of devices (e.g. cell phones, laptops and servers) presents a major challenge for programmers seeking to utilize their potential performance. Parallel programming has been extensively studied in the scientific computing community, resulting in a large body of knowledge. However, due the proliferation of multi- and many-cores, many programmers with little or no parallel programming experience are now encouraged to write parallel codes for multi-core devices. Tools to aid programmers to utilize the resources of multi-cores and thereby achieve high performance are therefore becoming increasingly important. Such tools should not only identify poorly performing code segments, but should also provide high-level, accessible information allowing the users to easily determine what is causing the poor performance. Furthermore, this type of information can also be valuable for computer architects in order to analyze the resource requirements of multi-threaded workloads.

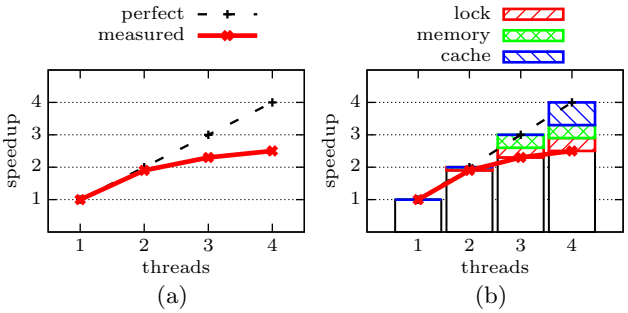


Figure 1: Speedup graph (a) and corresponding speedup stack (b).

Speedup Graphs: 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 1(a). 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. [7] identified the following as the 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 1(a) does not achieve perfect (linear) speedup. For example, it only achieves a speedup of 2.5 with four threads.

Speedup Stacks: 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. [7] introduced the concept of the speedup stack, illustrated in Figure 1(b). The speedup stack reports how much each of the scalability bot-

tlenecks (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 1(b) 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. [12] and Eyerman et al. [7] 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.

Contributions: In this paper we present a software-only profiling method for obtaining speedup stacks on commodity multi-cores. The method targets symmetric, fork-join style parallel programs with limited data sharing. This include 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. In this paper, we show how to obtain speedup stacks for a large range of applications from SPEC CPU2006 [13], PARSEC [1], NAS [17] and SPLASH-2 [20] using our software-only profiling method. Furthermore, we discuss how to interpret these speedup stacks and outline how the identified scalability issues can be resolved by both software and hardware developers.

Method: 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. Our approach to obtain speedup stacks therefore relies on the ability to emulate this hypothetical machine on commodity multi-cores.

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 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. 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 [5]. 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.

User Scenario: In order to optimize the scalability of a multi-threaded program, we envision speedup stacks to be used as follows. The first step is to find the parallel phases where the program spends most of its time. This can be done using tools such as Gprof [11], Oprofile [16] and VTune [15]. Out of these parallel phases, the one with the worst speedup is the most likely to reap the largest benefits from scalability optimizations. The second step is therefore to find this parallel phase. This can be done by analyzing the speedup graphs of the parallel phases. Once the most promising parallel phase has been identified, its speedup stack is used to determine which of the scalability bottlenecks are responsible for limiting the speedup. This information can then be used to determine what type of optimizations to apply and the upper bound on the expected improvements.

Outline: The rest of this paper is organized as follows. We begin by examining the concepts of the speedup stack (Section 2). This is followed by a review of Cache Pirating (Section 3). As a first step towards developing our method to obtain speedup stacks for multi-threaded programs, we show how to obtain speedup stacks for multiple co-running instances of single-threaded programs (Section 5). Then, we present our profiling framework for multi-threaded programs (Section 6.1). At this point we are equipped to present how we obtain speedup stacks for multi-threaded programs, first, for programs without synchronization (Section 6.2) and, then, for programs with synchronization (Section 6.3). Finally, we discuss related work (Section 7) and conclude (Section 8).

2. SPEEDUP STACKS

In this section we examine the key concepts of the speedup stack. We leave out the effects of Amdahl's law which explains the impact of the non-parallelizable phases on multi-threaded performance and instead focus only on the parallel phases[1]. However, including the effects of Amdahl's law in the speedup stack is a simple exercise. For clarity we lump together the scalability bottlenecks into three categories: 1) contention for shared cache capacity; 2) contention for off-chip memory resources, e.g. DRAM banks, data and address busses; and 3) synchronization, which includes lock contention, barrier synchronization and load imbalance.

[1]For a fork-join parallel program, a parallel phase occurs between a fork and the corresponding join.

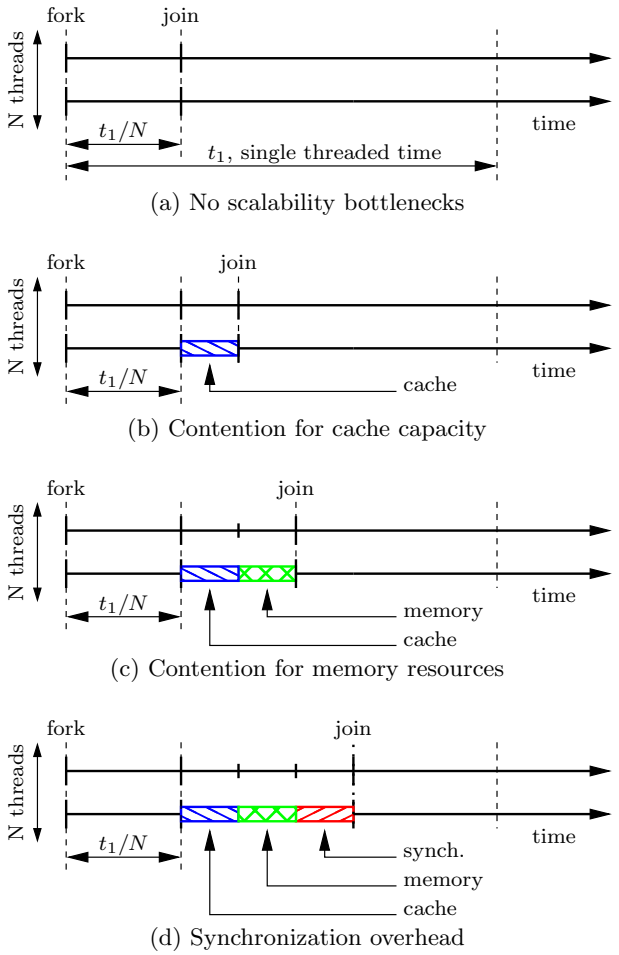


Figure 2: The execution of a parallel phase in four scenarios. For each scenario, a new scalability bottleneck is added to the hypothetical machine.

In this work we define the speedup in terms of the speedup that a parallel phase would achieve when run on a hypothetical machine where 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.

Figure 2 shows the execution time of a parallel phase on such a hypothetical machine with N cores in four different scenarios. In the top scenario the hypothetical machine has no scalability bottlenecks, and the parallel phase therefore achieves perfect (linear) speedup. For each successive scenario, a new scalability bottleneck is introduced. In the bottom scenario all scalability bottlenecks are present, and the parallel phase therefore achieves the same scaling as it would on the real machine. As the execution time of the parallel

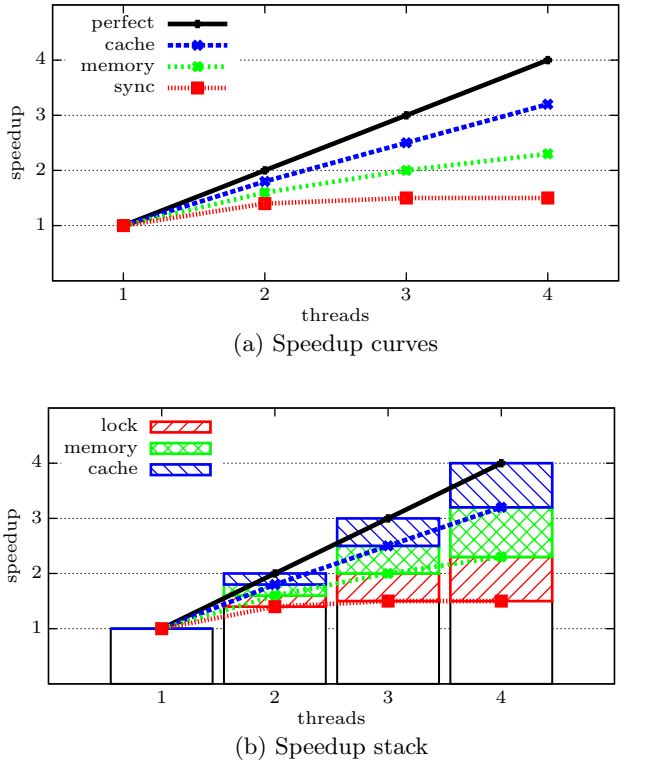


Figure 3: Speedup curves (a) and the corresponding speedup stack (b).

phase increases in each successive scenario, the speedup decreases (gets worse). Figure 3(a) shows the speedup curves achieved in the four scenarios. We define each speedup stack component as the difference between speedup curves achieved in the scenarios before and after the corresponding bottleneck was introduced. This is shown in Figure 3(b).

There are several options for how to define the speedup stack. Which one is preferred depends on how it is to be used. Defining the speedup stack in terms of the speedups on a hypothetical machine brings the question about the order in which to introduce the bottlenecks. While this order is largely arbitrary, we argue that the memory component of a speedup stack should be measured in a scenario where threads contend for cache capacity. The contention for off-chip memory resources depends on the threads' demands for memory bandwidth. Generally, when the cache capacity received by the threads is reduced (due to contention), their demand for off-chip memory resources increases. As a consequence their sensitivity to contention for off-chip memory resources increases. Therefore, in order to not underestimate the memory component of the speedup stack, it is important that the slowdown due to contention for off-chip memory resources is assessed in a scenario where the threads contend for cache capacity.

3. CACHE PIRATING

Cache Pirating [5] is an accurate profiling method for measuring application performance, such as execution rate (IPC) and off-chip bandwidth (GB/s), as a function of the amount of shared cache capacity available to the profiled application.

The basic idea of Cache Pirating is to control how much

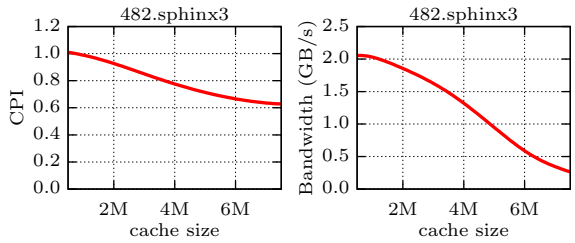


Figure 4: CPI (left) and Bandwidht (right) as a function of cache size of 482.sphinx3.

shared cache capacity is available to the profiled application (the Target) by co-running it with a cache-"stealing" application (the Pirate). The Pirate steals cache from the Target by ensuring that its working set is always resident in the shared cache. This effectively reduces the cache capacity available to the Target. As long as the Pirate keeps its working set in the cache, the Target can utilize the remaining cache capacity. By monitoring the Target using hardware performance counters while co-running it with the Pirate, we can capture any available performance metric as a function of how much of the shared cache capacity is available to the Target.

How successful the Pirate is stealing cache depends on the Target and how much it fights back. However, using performance counters we can readily determine if the Pirate is successful in stealing the requested amount of cache. When the fetch ratio of the Pirate is zero, we can be sure its entire working set is resident in the cache, since none of its data is fetched from main memory. Monitoring the fetch ratio of the Pirate therefore allows us to determine whether the Pirate successfully "steals" the desired amount of cache. If the Pirate's fetch ratio reaches above a threshold we know that it cannot "steal" the requested amount of cache capacity and we are forced to discard the measurement. Importantly, a low Pirate fetch ratio also ensures that the Pirate does not consume any of the shared off-chip memory resources.

Figure 4 shows data captured using Cache Pirating for 482.sphinx3. This data was obtained with a single profiling execution of the Target application, and show the Target's CPI (Figure 4(a)) and Bandwidth (Figure 4(b)), as a function of the cache capacity it receives. Since we ensure that the Pirate has a virtually zero fetch ratio, it does not consume any off-chip memory resources. As a result the Target gets exclusive access to all off-chip memory resources during profiling. Figure 4(a) therefore reports the CPI that the Target achieves when it is not slowed down due to contention for off-chip memory resources. Furthermore, Figure 4(b), shows the bandwidth consumed by the Target in this scenario, and can therefore be interpreted as the Target's bandwidth demand. If the Target receives significantly less than this bandwidth, due to contention, it will execute with CPI higher than that reported in Figure 4(a).

4. EXPERIMENTAL SETUP

The experiments presented in this paper have been run on a dual socket, quad core Intel Xeon E5520 (Nehalem) machine. Its cache configuration is detailed in the following table:

L1 cache	32k/32k, 8/4 way, inst./data, private
L2 cache	256k, 8 way, unified, private
L3 cache	8MB, 16 way, inclusive, shared

The memory controllers in this system have three memory channels each. Our baseline setup uses six dual-ranked 2GB DDR3-1333 DIMMs. For experimenting with only one active memory channel per socket we use two 4GB DDR3-1333 DIMMs, one for each socket.

The table below lists the single-threaded benchmarks used in this paper. These benchmarks were selected on the basis that they have the worst aggregate throughput when co-running four instances on our quad core Nehalem machine.

benchmark	suite	input
403.gcc	SPEC2006	166.i
433.milc	SPEC2006	su3imp.in
462.libquantum	SPEC2006	1397
470.lbm	SPEC2006	100_100_130_ldc
471.omnetpp	SPEC2006	omnetpp.ini
473.astar	SPEC2006	BigLakes2048.cfg
482.sphinx3	SPEC2006	ctlfile, args.an4

The table below lists the multi-threaded benchmarks used in this paper. A few of these benchmarks where originally parallelized with OpenMP. However, as our profiling framework currently only supports programs explicitly parallelized with pthreads, we ported these benchmarks to pthreads.

benchmark	suite	input
blackscholes	PARSEC	in_10M.txt
swaptions	PARSEC	512 swaptions
470.lbm	SPEC2006	100_100_130_ldc
DS	NAS	class D
IS	NAS	class C
fluidanimate	PARSEC	in_500K.fluid
ocean	SPLASH2	default

5. SINGLE-THREADED SPEEDUP STACKS

In this section we introduce our method to obtain speedup stacks. As a first step we look at how to obtain speedup stacks for multiple co-running instances of single-threaded applications. 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 components – *cache* and *memory*.

403.gcc: As a first example we show how to obtain the speedup stack for one, two and four co-running instances of 403.gcc. The curve labeled "measured" in Figure 5(a) shows the normalized throughput of 403.gcc on our quad core Nehalem machine with one memory channel activated. It shows that 403.gcc achieves far from perfect (linear) scaling. The curve labeled "pirate" shows 403.gcc's normalized throughput estimated based on the CPI measured using Cache Pirating.

To estimate the "pirate" curve, consider Figure 5(b) which reports 403.gcc's CPI obtained using Cache Pirating. Since

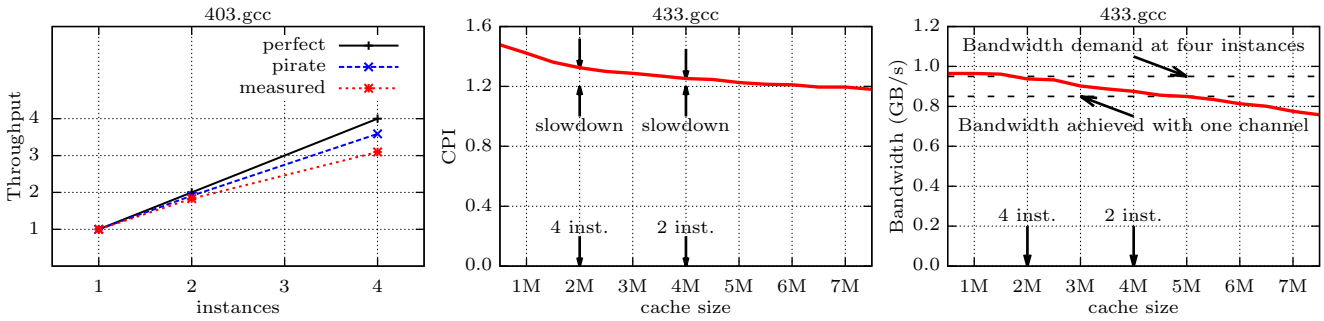


Figure 5: Speedup graph (a), CPI (b) and Bandwidth (c) as a function of cache size for 403.gcc.

each of the co-running instances execute the same code, they have the same demand for cache capacity, and consequently receive an equal amount. Each instance therefore gets 8MB, 4MB, and 2MB when co-running one, two and four, respectively. This allows us to find the CPI of the co-running instances from Figure 5(b). The CPIs are 1.18, 1.25 and 1.32, which corresponds to aggregate throughputs (IPCs) of 0.85, 1.60 and 3.03 when co-running one, two and four instances, respectively.

The throughput estimated above ("pirate") is clearly higher than the measured throughput ("measured"). See Figure 5(a). This is because the CPIs in Figure 5(b) are measured in a scenario where 403.gcc has exclusive access to the off-chip memory resources. Recall that while the Pirate "steals" cache capacity it does not consume hardly any off-chip memory resources. The aggregate throughput estimated using Cache Pirating therefore represents throughput that would be achieved in a scenario where each co-running instance competes for cache capacity, but have exclusive access to the off-chip memory resources. The difference between the curves labeled "perfect" and "pirate" in Figure 5(a) therefore estimates the speedup stack component representing the slowdown due to contention for cache capacity.

However, when actually co-running several instances they share the off-chip memory resources, which can cause significant slowdowns. This is the case for 403.gcc. The difference between the curves labeled "pirate" and "measured" in Figure 5(a) therefore estimates the speedup stack component that represents the slowdown due to contention for off-chip memory resources. Figure 6(a) shows the speedup stacks of 403.gcc when running on our quad core Nehalem machine, with one active memory channel. It indicates that the relative throughput of four co-running instances can be improved by about 0.4 if the cache bottleneck is resolved and by about 0.5 if the memory bottleneck is resolved.

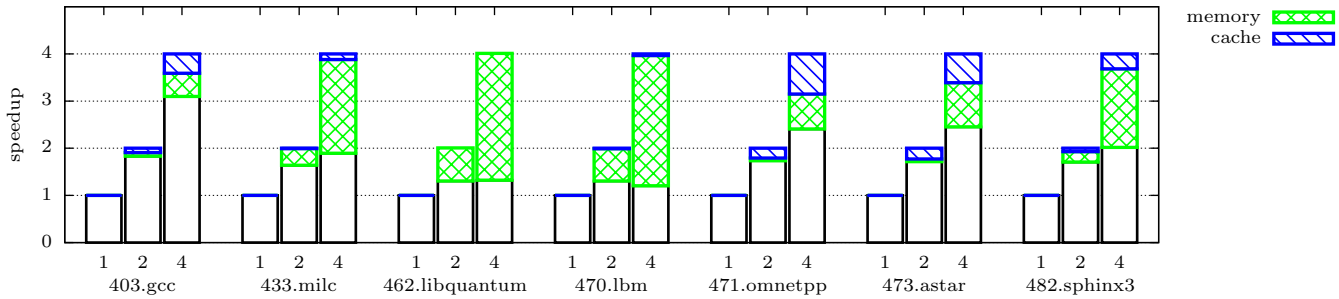
To further investigate 403.gcc's slowdown due to contention for off-chip memory resources we turn to Figure 5(c). It shows 403.gcc's bandwidth demand as a function of how much shared cache capacity it receives. When 403.gcc's cache capacity is reduced its bandwidth demand increases. For example, with 4/2MB cache available, 403.gcc's bandwidth demand is 0.85/0.95 GB/s. This corresponds to the bandwidth demand of each individual instance when co-running 2/4 instances. The figure also shows the bandwidth actually achieved by one instance when co-running four instances, which is slightly less than 0.85GB/s. The aggregate bandwidth demand (4×0.95GB/s) is therefore significantly higher the bandwidth actually achieved (4×0.85GB/s). This indicates that the memory system cannot deliver the aggregate bandwidth required by the four co-running instances of 403.gcc in order to achieve the CPI in Figure 5(b).

In this setup, with only one active memory channel, we can eliminate the off-chip memory resource bottleneck experienced by 403.gcc simply by increasing the number of active memory channels and thereby increasing the available off-chip memory resources. Figure 6(b) shows the speedup stack for 403.gcc with three active memory channels, the off-chip memory resource component is zero, indicating that 403.gcc no longer experiences any slowdown due to contention for off-chip memory resource. Indeed, measuring the aggregate bandwidth demand of four co-running instances of 403.gcc reveals that the memory system now has no problem meeting their bandwidth demand. Importantly, this is correctly reflected in the speedup stack.

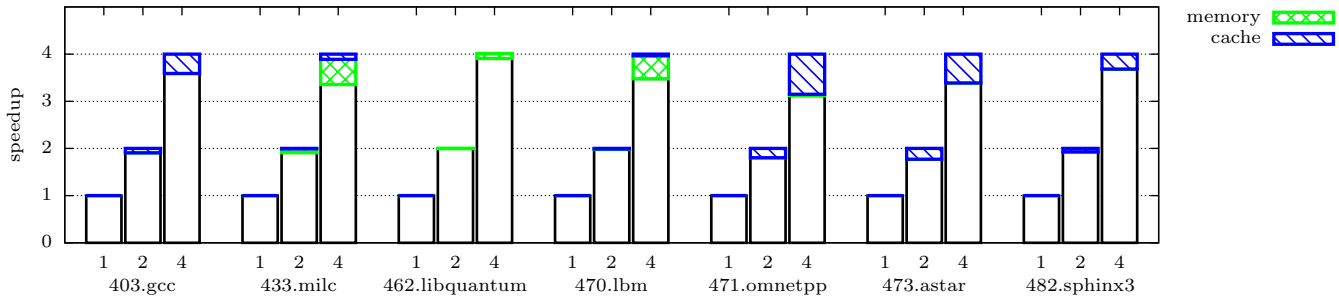
Figure 6 shows the speedup stack for seven SPEC CPU2006 benchmarks, for one (a) and three (b) active memory channels, respectively. Their speedup stacks indicate that their relative throughputs are limited by different scalability bottlenecks and to largely different degrees. For example, Figure 6(b) shows that 433.milc, 470.lbm and 473.astar all achieve relative throughputs in the range: $3.4 - 3.5$. However, in the case of 473.astar this is because of contention for cache capacity, while for 433.milc and 470.lbm it is because of contention for off-chip memory resources. As this cannot be established looking only at their speedup graphs, the observations above highlight the value the speedup stack.

Several of the applications, namely 403.gcc, 471.omnetpp, 473.astar and 482.sphinx3, have relatively low off-chip memory bandwidth demands[2]. With three memory channels, their bandwidth demands can be easily satisfied, and we therefore expect the components of their speedup stacks representing contention for off-chip memory resources to be zero. This allows us to assess the accuracy of our method to obtain speedup stacks. To predict a zero memory component, the relative throughput estimated using Cache Pirating should match the measured relative throughput. This

[2]Their bandwidth demands are 3.8GB/s, 2.4GB/s, 1.6GB/s and 3.5GB/s, respectively. These demands, however, do not correlate with their respective memory components in Figure 6(a). This is because different applications have different degrees of sensitivity to contention for off-chip memory resources [6].



(a) One active memory channel



(b) Three active memory channels

Figure 6: Speedup stacks of seven SPEC CPU2006 benchmarks.

is indeed the case. The largest relative difference between the "measured" and "pirate" curves is 5% (for 482.sphinx3). For our purposes, which is mainly to determine which of the components are the most significant, 5% is a small error.

In the remainder of this section we present analysis of 471.omnetpp's and 470.lbm's speedup stacks and discuss what both software developers and computer architects can do in order to improve their aggregate throughput.

471.omnetpp: With three active memory channels, 471.omnetpp's throughput is mainly limited by contention for cache capacity (see Figure 6(b)). There are two options to improve its 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. Consider the case of four co-running instances. If each instance gets 6MB of cache, their CPIs would be about 1.73, see Figure 7. This corresponds to a relative aggregate throughput of 3.93 ($1.70/1.73\times4$). 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.

470.lbm: Even with three active memory channels, 470.lbm's throughput is limited by contention for off-chip memory resources (see Figure 6(b)). To understand how to improve its throughput we turn to Figure 8 which shows 470.lbm's CPI and bandwidth captured using Cache Pirating. As its CPI is virtually flat, 470.lbm is unaffected 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. This is mainly due to 470.lbm's prefetch friendly access pattern. Figure 8(b) shows that the 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. Therefore, to improve its throughput there are two options, either increase the available off-chip memory bandwidth or reduce the bandwidth demand. First, 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 8(b) 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 a aggregate demand of 10.8GB/s ($4 \times$ 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.

Tackling a scalability bottleneck, either by increasing data locality, the cache capacity or the off-chip memory resources, might impact how the application is affected by the other bottlenecks. As we will see later, reducing the synchronization component, by, for example, reducing lock contention will increase the threads' execution rates. This, however, might increase the bandwidth demand of the application, and consequentially increase its memory component. However, attacking either of the bottlenecks considered in this section, contention for cache capacity and off-chip memory resources, does not negatively impact how the application is affected by the other bottleneck. First, increasing data locality or the cache capacity, in order to reduce the cache component, can, as a side effect, reduce the off-chip memory bandwidth demand and thereby *reduce* the memory component. Second, increasing to off-chip memory resources, in order to reduce the memory component, does not negatively impact the cache component. This is because an program's data locality is determined by the application's access pattern, and is therefore not impacted by the execution rate.

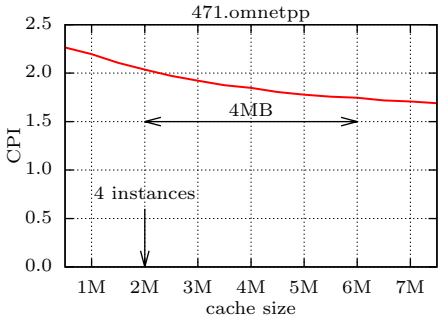


Figure 7: 471.omnetpp's CPI as a function of cache size.

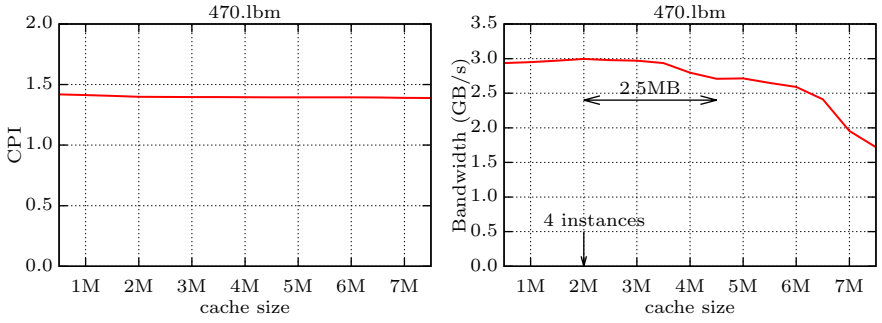


Figure 8: 470.lbm's CPI (a) and Bandwidth (b) as a function of cache size.

6. MULTI-THREADED SPEEDUP STACKS

In this section we present how to obtain speedup stacks for multi-threaded applications. These speedup stacks consist of three principal components: contention for shared cache capacity; contention for off-chip memory resources; and synchronization, which includes lock contention, barrier synchronization and load imbalance. To clarify the presentation, we first describe how to obtain speedup stacks for multi-threaded applications that do not use any synchronization operations (Section 6.2). We then extend this method to handle applications that use both locks and barriers (Section 6.3). Fundamental to this is a profiling framework that allow us to emulate a scenario where the profiled thread has exclusive access to all shared resources and where synchronization operations are for free (Section 6.1). To introduce and controlled scalability bottlenecks in this framework, we combine it with Cache Pirating.

In this work, we target symmetric, fork-join style parallel applications, where all threads execute the same code and have minimal data sharing. This includes many data parallel applications. For this class of applications all threads receive equal shares of the shared cache capacity (they execute the same code and therefore have the same memory demands) and we can therefore easily determine how much cache capacity each thread receives. This allows us to treat these multi-threaded applications in much the same way as we did co-running single-threaded applications. In order to handle asymmetric applications, we could use techniques, such as cache sharing models [2, 3], to determine how much cache capacity each thread receives. This, however, is outside the scope of this paper.

6.1 Profiling Framework

Our approach to emulate a profiling environment where the profiled thread has exclusive access to the shared resources and where synchronization is for free is conceptually very simply. We simply let the profiled thread run alone (i.e., no co-runners) on the system. This is achieved as follows.

Method: We launch the Target application with a given number of threads. When it reaches the fork-point of the first parallel phase of interest, we halt all but one thread, the one to be profiled, and let it run until it reaches the join or a synchronization barrier. To maintain correctness we need to carefully handle synchronization barriers. When the profiled thread reaches a synchronization barrier we halt both the profiled thread and let the other threads catch up. When the remaining threads reach the barrier, we only let the profiled thread through, and repeat the above process until a join is finally reached. This is illustrated in Figure 9.

To introduce Cache Pirating into the framework, we simply co-run the profiled thread with the Pirate application. The Pirate is always started and stopped in sync with the profiled thread. However, before starting the profiled thread after a barrier, we let the Pirate warm up its working set. Furthermore, in order to ensure that the Pirate "steals" the desired amount of cache we monitor its off-chip fetch ratio.

Features: With this setup the profiled thread will not experience any lock contention. Since it is not running at the same time as the other threads, none of the other threads will hold any locks while the profiled thread is running (they are waiting at a barrier). The profiled thread will therefore always get immediate access to the locks without having to spin or sleep. Furthermore, the profiled thread will have exclusive access to the off-chip memory resources, while at the same time we can control exactly how much cache capacity it receives.

Implementation: The framework is implemented as a dynamic loadable library which overloads `pthread_create`, `pthread_join` and `pthread_barrier_wait`. Preloading the library when launching the Target allows us to take control of the threads, and start and stop them at the fork, join and barriers as described above. When intercepting calls to `pthread_create` at a fork-point, the library attaches a set of performance counters to the profiled thread. When intercepting a call to `pthread_barrier_wait`, the first thing the library does is to read the performance counters. It reads the counters again right before the profiled thread leaves the barrier. By recording the difference between the counters read right after and right before entering and leaving the barrier, the measurements do not include the time the profiled thread spends waiting at barriers. In effect the performance counter measurements will look as if the profiled thread never had to wait at barriers. At the same time we carefully monitor the Pirate's fetch ratio to ensure that it is capable to steal the desired amount of cache and discard the measurements when its fetch ratio is too high.

Limitations: Our current implementation does not handle condition variables and have only been tested with applications that are explicitly parallelized with pthreads. However, we believe that it can be extended to handle OpenMP applications on Linux, since GOMP, GCC's OpenMP implementation, uses pthreads internally. Alternating between running the profiled thread alone and running the other threads, has several consequences. The data cached by the

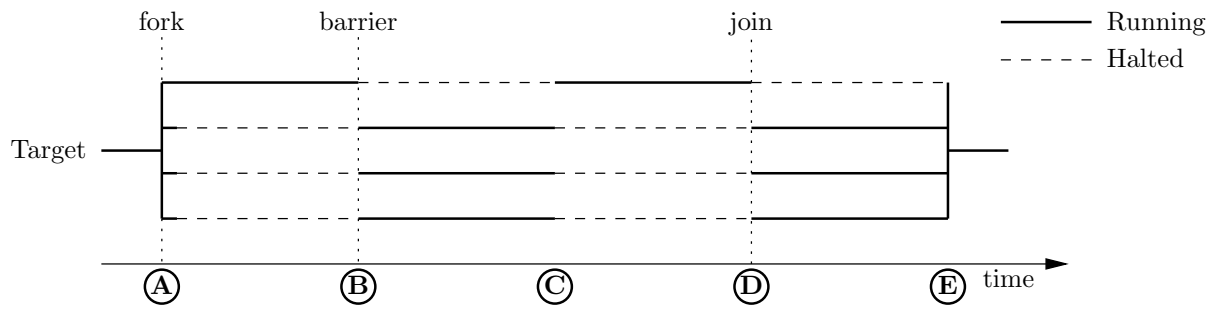


Figure 9: Chart showing how the profiling framework works. (A) Halt all but the profiled thread. (B) The profiled thread reached a barrier, start the other threads and let them catch up. (C) At this point, all threads have reached the barrier. Repeat A until the join is reached. (D) The profiled thread reached the join, halt it and let the other threads catch up. (E) All threads have made it through the parallel phase. Profiling done.

profiled thread might be evicted by the other threads; this might not have happened had they all run together. The impact of this is most pronounced when the time between barrier synchronizations is relatively short. To work around this issue, we pin the non-profiled threads to run on a different socket so as to not share cache with the profiled thread. For applications whose threads share data, the threads can "prefetch" data for each other. As the profiled thread runs alone, it will not directly experience such data "prefetching". However, as we target applications with limited data sharing these issues are largely avoided.

6.2 Applications without Synchronization

In this section we show how to obtain speedup stacks for multi-threaded applications that do not use any synchronization operations, e.g. barriers and locks. For such applications there are two main scalability bottlenecks – contention for cache capacity and off-chip memory resources. There is however a third scalability bottleneck – cache coherency, which can cause cache lines in the private caches (L1 and L2, on our Nehalem machine) to be invalidated. This, in turn, can cause additional cache misses and ultimately impact performance. The method presented in this paper does not handle cache coherency. However, we are mainly concerned with applications that have limited data sharing and therefore require limited coherency activity. Furthermore, the penalty of private cache misses can often be effectively reduced by out-of-order execution [9]. Therefore, the cache coherency component of these applications are much less significant, and we do not include it in the speedup stacks.

As the speedup stacks for data parallel applications without synchronization consist of the same components as the speedup stack for a set of co-running instances of a single-threaded application, we can handle them in much the same way. Figure 10 shows three speedup curves of the most significant parallel phases of IS (integer sort). The ones labeled "perfect" and "measured" report the perfect (linear) speedup and the speedup measured on our quad core Nehalem machine, respectively. To estimate the "pirate" speedup curve, we launched IS under the control of profiling framework presented in Section 6.1 with one, two and four threads, and used Cache Pirating to limit the profiled thread's cache capacity to 8MB, 4MB and 2MB, respectively. This effectively

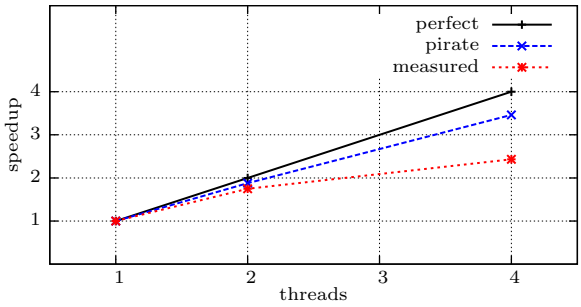


Figure 10: Speedup graph of IS.

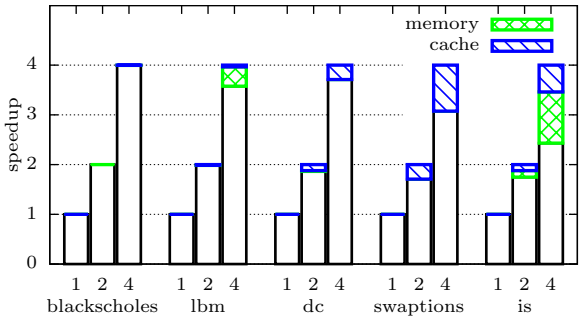


Figure 11: Speedup stacks for five multi-threaded applications without synchronization operations.

emulates a scenario where the profiled thread experiences contention for cache capacity but gets exclusive access to all off-chip memory resources. Figure 11 shows the speedup stack of the most significant parallel phase of IS (on the far right). Its cache and memory components are computed as the differences between the speedup curves labeled "perfect" and "pirate"; and "pirate" and "measured", respectively.

Figure 11 shows the speedup stack for five fork-join style data parallel programs that do not use any synchronization operation. These programs have a wide range of speedup behaviors and largely different speedup stacks. To get an intuitive sense of what is causing the sizes of their components and assess their accuracy, we outline the relevant qualitative properties of a few of the applications shown in Figure 11 and discuss how they impact the speedup stacks.

Blackscholes accesses several large arrays that are sliced

up and distributed among its threads. However, it has very limited data locality, only a few temporary variables stored on the stack are ever reused. We therefore expect its cache component to be zero. While its aggregate working set does not fit in the last-level cache, its computational intensity (instructions per off-chip memory access) is high and its aggregate off-chip memory bandwidth demand is therefore low, less than 0.5GB/s. This demand can easily be satisfied and we therefore expect its memory component to be zero.

IS sorts a set of integer keys. In its most significant parallel phase, it inserts the keys into a histogram. The keys are first distributed among the threads, such that each thread gets the number of keys. The threads then insert its subset of the keys into its own private histograms. These histograms are finally merged at the end of the parallel phase[3].

The keys are only read once, and are therefore never reused. As a result all keys are fetched from off-chip memory. Since the operation of inserting the keys into histograms is performed in a tight loop, IS has a larger aggregate demand for off-chip memory bandwidth (15GB/s) than can be satisfied. Consequently, it has a large memory component.

With the input parameters used above, the same key values reoccur many times in the set of keys to be sorted. Each histogram bucket is therefore accessed multiple times and in random order, resulting in a fair amount of data locality. However, the histogram sizes are independent of the number of threads, and the total amount of cache required to hold the most recently used buckets therefore increases with the number of threads. This causes the number of cache misses to increase when the number of threads are scaled up and contributes both to the cache and memory components.

Swaptions is similar to IS, its input data is distributed among the threads, and its threads have fairly large private working set whose size is indifferent to the number of threads. However, similarly to Blackscholes, its computational intensity is low and its aggregate bandwidth demand (2.6GB/s) can easily be satisfied. Hence, the large cache component and the zero memory component.

Lbm's speedup stack is very similar to its single-threaded counterpart. (See Figure 6.) This is expected. Lbm performs a stencil computation over a three dimensional matrix. In the serial version, it sweeps each plane of the matrix top to bottom and accesses each element in the plane along with its two immediate neighbors in all six spatial directions. As the stencil "prefetches" the two planes below the currently processed, there is a fair amount of data reuse. In the single-thread case, the speedup stack in Figure 11, suggests that there is very little slowdown due to cache contention when co-running four instances. This is expected. In the case of four co-running instances, the last-level cache can hold 12+ planes, and can therefore cache three planes for each instance, this allows them to turn the data reuse across planes into cache hits. Lbm is parallelized by giving each thread a range of consecutive planes to process. This makes the data locality of the multi-threaded version similar. In both cases increasing the number of instances or threads will not increase the number of cache misses. However, when co-running four instances, each will have their own copy of the matrix, while in the multi-threaded case there will only be one copy. But, since the last-level cache cannot even hold a single copy of the matrix, there will be no extra cache misses when increasing the number of threads/instances. Hence, we expect the cache component to be zero in both cases, which indeed it is. In terms of off-chip memory bandwidth, each thread/instance has the same demand which is independent of the number of threads/instances, and we therefore expect the memory components to be the same in the two cases.

[3]This requires synchronization, but as we are interested in application without synchronization in this section, we moved the merging code to after the parallel phase.

6.3 Applications with Synchronization

In this section we extend the method presented in the previous section to include a synchronization component in the speedup stacks. This component includes slowdowns due to lock contention, barrier synchronization and load imbalance. In Section 2, we defined this component in terms of the speedup measured in a hypothetical environment in which synchronization is for free, i.e. the threads never have to wait, spinning or sleeping, at locks and barriers. However, rather then emulating such an environment, we simply measure the time each thread spends waiting for access to critical sections protected by locks and the time waiting at barriers when the program is running on the real machine. We then use these measurements to estimate the speedup that would be achieved in a scenario where synchronization is for free.

To measure the time a thread is spinning, waiting for access to a lock, we overload the pthread functions related to mutex and spinlock operations. The `pthread_mutex_lock` is overloaded with a function that reads the time stamp counter, using the `rdtscp` instruction, both before (tsc_1) and after (tsc_2) it calls the real `pthread_mutex_lock`. The time spent waiting for access to the lock is then $tsc_2 - tsc_1$. (Both `pthread_spin_lock` and `pthread_barrier_wait` can be handled the same way.) By accumulating these differences for each thread individually, we get the total time they spend waiting for access to locks.

Ultimately we want to estimate how much faster a parallel phase would execute if all synchronization operations were for free. We consider two cases, parallel phases with and without synchronization barriers. First, for parallel phases without barriers, we measure each threads execution time (t) and the time it spends spinning or sleeping to acquire locks (t_l). In a scenario where the threads do not have to wait to acquire locks, their execution times would be equal to $t - t_l$. However, the execution time of a parallel phase is equal to that of the slowest thread. Therefore, for parallel phases without barriers, the execution time when lock synchronization is for free equals $t - t_l$ of the slowest thread, i.e., the one with the largest $t - t_l$. Second, for parallel phases with barriers, the synchronization overhead consists of two components: overhead incurred due to lock synchronization and barrier synchronization. While our simple profiling method (outlined above) can easily be extended to provide the information needed to distinguish between the two components, we merge them into single component for the purpose of building speedup stacks. This allows us estimate the synchronization component of the speedup stack, simply by measuring the total time each thread spend waiting for both locks and barriers (t_{l+b}), and subtract it from the threads' measured execution times (t). As above, the execution time of the parallel phase is equal to that of the slowest thread, in this case the one with the largest $t - t_{l+b}$.

To account for load imbalance, we treat the join operation at the end of the parallel phase as a barrier. This allows us to

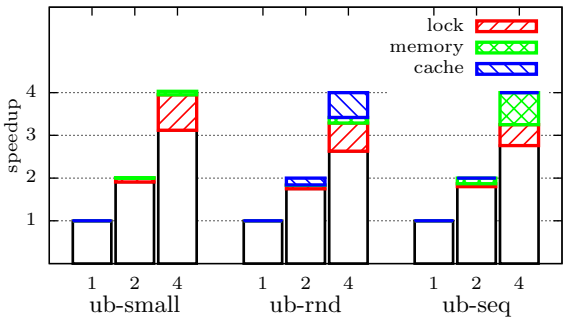


Figure 13: Speedup stacks of three micro benchmarks.

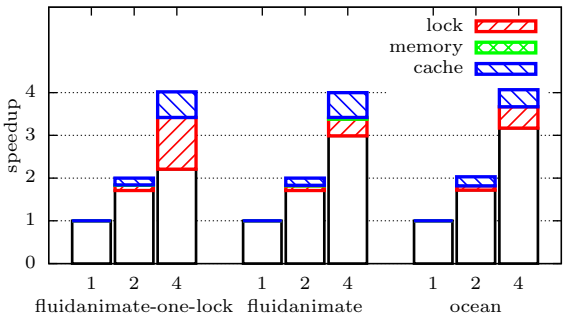


Figure 14: Speedup stacks for three multi-threaded programs with synchronization operations.

handle lock and barrier synchronization; and load imbalance in a uniform way. Our speedup stacks therefore report one component – *synchronization*, which include all the three synchronization overheads.

```
while (1) {
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            compute(iterations);
            pthread_mutex_lock(&mutex);
            // Short critical section
            pthresd_mutex_unlock(&mutex);
        }
        pthread_barrier_wait(&barrier);
    }
}
```

Listing 1: A common synchronization pattern

To assess the above method we start by evaluating it using three simple micro benchmarks that replicate a common synchronization pattern of data parallel applications. Each thread of the micro benchmarks runs the code in Listing 1. The three micro benchmarks have their own `compute` function, in which they perform computations and access memory. The `iteration` parameter determines the amount of computations performed by the `compute` function. Furthermore, the threads do not share any data, they access disjoint parts of the working set, and will therefore execute correctly without synchronizing. This allows us to obtain references for evaluating the method presented in this section by measuring the micro benchmarks' speedups with the synchronization primitives commented out.

Figure 12(a) shows five speedup curves for the first micro benchmark, called ub-tiny, where the curves labeled "perfect", "pirate" and "measured" represent the same speedups as in previous sections. The curves labeled "w/o synch. estimated", is the speedup estimated using the above method when both locks and barriers are for free, and the curve labeled "w/o synch. measured" is the speedup measured on our Nehalem machine when all synchronization operations are disabled (i.e. commented out). The working sets of the threads in this micro benchmark fit in the L1 caches, and its speedup is therefore not limited by contention for either cache capacity or off-chip memory resources. The "pirate" curve matches the "perfect" and the cache component of the speedup graph (see Figure 12) is therefore zero as expected. The "w/o synch. estimated" and "w/o synch. measured" also match almost perfectly. This means that for the ub-tiny micro benchmark the above method almost perfectly estimates the speedup when synchronization operations are for free. Finally, the synchronization component of the speedup stack is the difference between "'w/o synch. estimated" and "measured". As expected, the speedup stack shown in Figure 13 has only one significant component, synchronization. It has however a small memory component, this is because the "w/o synch. estimated" slightly underestimates the "perfect" curve.

Figure 12(b) and 12(c) shows the same data as Figure 12(a) for the other two micro benchmarks, ub-rnd and ub-seq. Both micro benchmarks have the following properties. Their threads access disjoint parts of their working sets; their working sets do not fit in the last-level cache; and their `iterations` parameters are set so that they suffer from lock contention. The speedup stacks for the two micro benchmarks are shown in Figure 13. Ub-rnd accesses its working set randomly. Its threads therefore have fair amounts of data locality and suffer from contention for cache capacity. Furthermore, its aggregate bandwidth demand is low enough that it can be easily satisfied. We therefore expect its lock and cache components to be significant while its memory component is zero. Ub-seq on the other hand accesses its working set sequentially at a high rate. Its threads therefore never hit in the last-level cache and are thereby not impacted by cache contention. However, it has a high demand for off-chip memory bandwidth which can not be satisfied. We therefore expect its lock and memory component to be significant while its cache component is zero.

For both benchmarks, the "w/o synch. estimated" and "w/o synch. measured" match very well (see Figure 12). This indicates that the method above for estimating the speedup when synchronization is free works fairly well. Furthermore, for ub-rnd, which is expected to have a zero memory component, the "w/o synch. estimated" and the "pirate" curve are expected to match up. This is indeed the case, which further confirms the accuracy of the method.

When performing these experiments, we swept the configurable parameters (N, M and iterations) over large ranges. For all parameters combinations, the prediction errors where very similar to those shown in Figure 13. The results in Figure 13 where selected because the speedup stack components are large and therefore clearly visible in the figures.

Figure 14 shows the speedup stacks, including synchronization components, for the three data parallel benchmarks with the largest synchronization overheads that we could find that do not use condition variables (which we currently

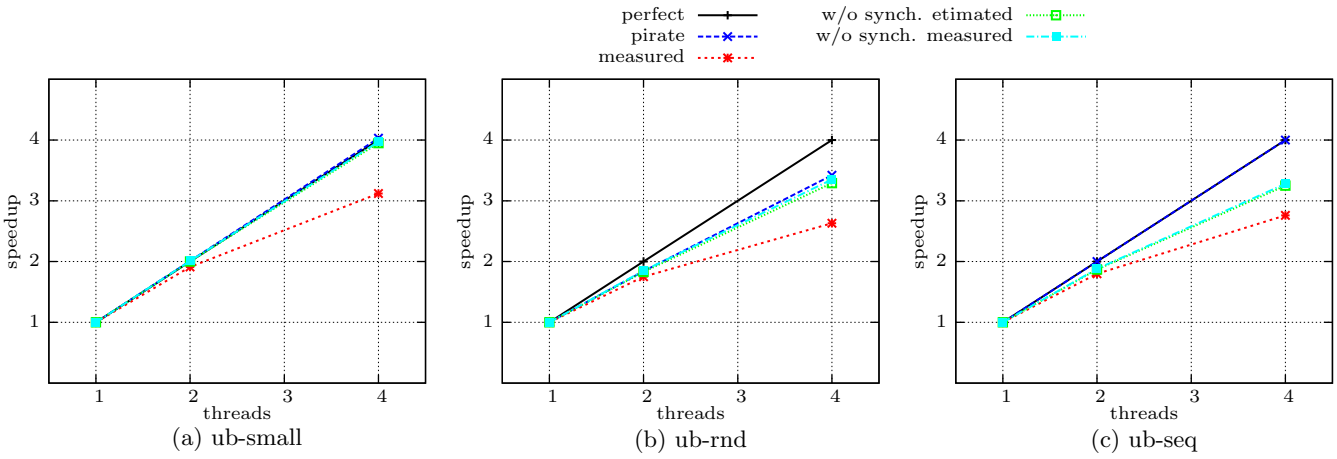


Figure 12: Speedup graphs of three micro benchmarks.

do not support). 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 (from PARSEC [1]), 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 neighbours. To introduce more lock contention we created a version of fluidanimate that uses a single lock (fluidanimate-one-lock).

There are several things to note. Figure 14 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 are 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 14 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.

7. RELATED WORK

Cycle Stacks: A cycle stack (a.k.a. CPI stack) breaks down a single-thread program's execution time into a number of cycle components which represent how many cycles the program is stalled due to various miss events, such as cache and TLB misses; and branch miss predictions. Cycle stacks are widely used by software developers and computer architects to gain high-level insight into the behaviors of applications [8]. We envision the speedup stack to be used in a similar manner to analyze multi-threaded programs. One of the major challenges when building cycle stacks is to correctly account for the overlapping of miss events [8, 10]. For example, the latency of a last-level cache miss can be "hidden" by that of another last-level cache miss. In this work, however, we do not count miss events, but rather build our speedup stacks by measuring how the profiled program's speedup is impacted when introducing the scalability bottlenecks one-by-one. This allow us to attribute the reduced speedup to the component representing the introduced bottleneck, and the overlap of miss events is therefore not a concern. This can make the speedup stack dependent on the order in which the bottlenecks are introduced. However, we argue that the order used in this paper produce natural and easy to interpret speedup stacks (see Section 2).

Speedup Stacks: Heirman et al. [12] propose to use cycle stacks to analyze multi-threaded programs. They simulate the analyzed program and capture cycle stacks for each individual thread. In order to analyze the profiled program's scaling behavior, they perform several simulations with increasing number of cores. While they do not present their data in the form of a speedup stack, applying their method to fork-join style multi-threaded programs produces data very similar to that typically reported in a speedup stacks.

Eyerman et al. [7] presents a hardware performance counter architecture for obtaining speedup stacks. They use the proposed counters to measure a set of performance metrics while the analyzed program is running on a (simulated) multi-core implementing their performance counter architecture. Based on these metrics, they estimate the execution rate that the analyzed application would achieve had it not been slowed down due to scalability bottlenecks. This allows them to estimate the slowdown due to individual scalability bottlenecks and build speedup stacks.

While both Heirman et al. [12] and Eyerman et al. [7] report encouraging results, the method presented in this paper does not require either simulations or custom hardware support, the speedup stacks are instead captured on real, commodity multi-cores.

Synchronization: There are several methods to measure the cost of synchronization [14, 19, 18]. These methods can, for example, track waiting times for individual locks

and threads [14]; find which threads are holding locks for excessive periods of time and thereby causing synchronization costs [19]; and measure the cost of load imbalance [18]. However, the overheads introduced when instrumenting the synchronization primitives can adversely impact the profiled applications performance and therefore perturb the measurements [4]. We therefore opted to use the simple, low-overhead method presented in Section 6.3 to measure synchronization overheads. However, in case that the program's scalability is limited by synchronization, the above methods can be used, after the speedup stack has been obtained, in order to figure out how to best reduce synchronization overheads.

8. SUMMARY AND CONCLUSIONS

In this work we present a profiling method for obtaining speedup stacks for symmetric, fork-join, data-parallel applications. To capture the profiling data used to construct the speedup stack, we present a profiling method that allows us to emulate a controlled environments on commodity multi-cores in which we can eliminate the selected scalability bottlenecks one-by-one. As the profiling data is captured using hardware performance counters on a real machine (i.e., not in a simulator), the speedup stacks account for all idiosyncrasies of the machine on which the application is profiled. The resulting speedup stacks reports three components: contention for cache capacity; contention for off-chip memory resources; and synchronization, which include lock contention, barrier synchronization and load imbalance. Importantly, these components represent the main scalability bottlenecks of the targeted class of programs.

To assess the accuracy of the obtained speedup stack we did the following. First, to evaluate the cache and memory components, we looked at applications that do not synchronize and whose off-chip memory bandwidth demands is low enough that it can be satisfied. We expect such applications to have zero sized memory components. This was indeed the case. The largest error was 5%, which indicate that our profiling method accurately estimates both the cache and memory components for these applications. Furthermore, we discussed the relevant qualitative properties of a set of applications and showed that the speedup stacks account for these properties as expected. Second, to investigate the accuracy of the synchronization components, we looked at applications that synchronize and whose memory components are zero. For such applications the cache and synchronization components should add up and together 'fill" the speedup stack. This was indeed the case for all applications we looked at. Together these experiments indicate accurate speedup stacks can be obtained on commodity multi-cores.

Speedup stacks imply which bottlenecks are the most beneficial to tackle in order to improve applications' scalability. However, they do not alone provide enough information to determine *how* to improve the application's speedup. The scalability of programs limited by contention for shared resources, e.g. cache capacity and off-chip memory resources, can be improved, either by increasing the capacity of the resource or by reducing the applications' demand for the resource. The profiling method presented in this paper captures additional data beyond that needed to build the speedup stack. This data can be used to determine both how much to increase capacity or reduce demand to achieve a given speedup. While we could easily extend our profiling method to capture basic information that would allow us to improve synchronization, several methods and tools have already been proposed for this purpose [14, 19].

9. REFERENCES

[1] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[2] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA*, 2005.

[3] X. E. Chen and T. M. Aamodt. A first-order fine-grained multithreaded throughput model. In *HPCA*, 2009.

[4] J. Demme and S. Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *ISCA*, 2011.

[5] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache pirating: Measuring the curse of the shared cache. In *ICPP*, 2011.

[6] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Bandwidth bandit: Understanding memory contention. In *ISPASS*, 2012.

[7] S. Eyerman, K. D. Bois, and L. Eeckhout. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *ISPASS*, 2012.

[8] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In *ASPLOS*, 2006.

[9] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comp. Sys.*, 27(2), 2009.

[10] S. Eyerman, K. Hoste, and L. Eeckhout. Mechanistic-empirical processor performance modeling for constructing CPI stacks on real hardware. In *ISPASS*, 2011.

[11] J. Fenlason and R. Stallman. GNU Gprof. `http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html`.

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

[13] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4), 2006.

[14] Intel. Parallel amplifier. `http://software.intel.com/en-us/articles/intel-parallel-amplifier/`.

[15] Intel. Vtune™ performance analyzer. `http://www.intel.com/software/products/vtune`.

[16] J. Levon. Oprofile. `http://oprofile.sourceforge.net`.

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

[18] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey. Scalable identification of load imbalance in parallel executions using call path profiles. In *SC*, 2010.

[19] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *PPoPP*, 2011.

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