# Methodes voor de prestatieanalyse van het schalingsgedrag van meerdradige programma's

# Performance Analysis Methods for Understanding Scaling Bottlenecks in Multi-Threaded Applications

Kristof Du Bois

Promotoren: prof. dr. ir. L. Eeckhout, dr. ir. S. Eyerman
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2013 - 2014

UNIVERSITEIT
GENT

ISBN 978-90-8578-703-7
NUR 980
Wettelijk depot: D/2014/10.500/49

*To my family.*

# Dankwoord

De weg naar een doctoraat is lang en verloopt over een parcours dat onderweg bezaaid is met verschillende hindernissen. Om deze tocht toch tot een goed einde te brengen, heb ik de afgelopen vier jaar de hulp en steun gekregen van een aantal bijzondere mensen. Daarom, nu de finishlijn van dit doctoraat bereikt is, is het dan ook de hoogste tijd om hier even bij stil te staan.

Allereerst wens ik mijn beide promotors, prof. Lieven Eeckhout en dr. Stijn Eyerman, oprecht te bedanken voor hun onvoorwaardelijke steun, inzichtvolle feedback en actieve hulp bij het schrijven van artikels en als apotheose deze thesis. Dat ik dit doctoraat tot een goed einde heb kunnen brengen, is voor een groot deel dankzij hen. Daarom overdrijf ik niet wanneer ik zeg dat ik mij geen betere begeleiding had kunnen voorstellen.

*I would also like to thank dr. Jennifer B. Sartor for all her great help on writing papers, her assistance with Java, and for providing interesting ideas.*

Daarnaast wil ik alle leden van mijn examencommissie bedanken, omdat ze de tijd namen om dit proefschrift te lezen, in vraag te stellen en te verbeteren. *I would also like to address a special word of thanks to Prof. Margaret Martonosi and Prof. Per Stenström for accepting my invitation to be part of my PhD committee and their willingness to travel to Ghent, despite their busy schedules.*

Wat ik bovendien nooit zal vergeten, is de aangename en vriendelijke werksfeer die heerst in onze onderzoeksgroep. Daarom wil ik alle collega's en in het bijzonder mijn bureaugenoten van de afgelopen jaren bedanken: Andy, Cecilia, Frederick, Max, Sam, Sander, Shoaib en Stijn. Ook Marnix, Michiel en Ronny mogen hier zeker niet ontbreken, onder meer omwille van hun zeer geapprecieerde hulp bij praktische problemen. Een heel speciaal woordje van dank heb ik voor Max en Klaas, twee collega's die al snel goede vrienden werden en waarmee ik ook

naast het werk tal van onvergetelijke momenten mee heb beleefd.

Het Vlaams Supercomputer Centrum (VSC) voor het beschikbaar stellen van een enorme hoeveelheid rekenkracht en de feilloze technische ondersteuning.

Ten slotte wil ik mijn familie en vrienden bedanken om er steeds te zijn wanneer ik hen nodig heb. In het bijzonder wil ik mama en papa bedanken, eigenlijk kunnen woorden niet beschrijven hoe dankbaar ik jullie ben omwille van alle kansen die jullie mij gaven en me steeds steunen in alles wat ik doe en me vormden tot de persoon die ik nu ben, enz.

Kristof Du Bois
Gent, 24 juni 2014

# Examencommissie

Prof. Rik Van de Walle, *voorzitter*
Decaan Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Dr. Jennifer B. Sartor, *secretaris*
Vakgroep ELIS
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Lieven Eeckhout, *promotor*
Vakgroep ELIS
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Dr. Stijn Eyerman, *promotor*
Vakgroep ELIS
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Filip De Turck
Vakgroep Informatietechnologie
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Wolfgang De Meuter
Vakgroep Computerwetenschappen
Faculteit Wetenschappen
Vrije Universiteit Brussel

Prof. Margaret Martonosi
Princeton University
USA

Prof. Per Stenström
Chalmers University
Sweden

# Leescommissie

# Samenvatting

Gedurende vele jaren konden processorfabrikanten de prestatie van één enkele rekenkern verhogen. Er was immers een toename in transistordensiteit, als gevolg van de wet van Moore; en de schalingstheorie van Dennard stelde dat het verbruikte vermogen van een transistor schaalde met de grootte ervan. Tijdens deze periode resulteerde de trend van dalende transistorgroottes in een toenemende klokfrequentie en/of pijplijnbreedte van de processor, waarvan de Intel Pentium 4 een goed voorbeeld was. Aan deze trend kwam echter een einde omwille van beperkingen op het verbruikte vermogen en de daaraan gerelateerde koeling. Als reactie hierop introduceerden processorfabrikanten multi-core processors waarbij meerdere rekenkernen op eenzelfde chip geplaatst worden.

De verschillende rekenkernen op een multi-core processor kunnen onafhankelijk van elkaar instructies uitvoeren, maar ze moeten wel een aantal componenten op de chip delen, zoals bijvoorbeeld een cache of een bus. Het delen van deze componenten zorgt ervoor dat ze efficiënter gebruikt worden, maar heeft als nadeel dat de verschillende rekenkernen een invloed hebben op elkaars prestatie. Bijvoorbeeld in het geval van een gedeelde cache kan een rekenkern ervoor zorgen dat data van een andere rekenkern verwijderd wordt, wat leidt tot een toegenomen aantal missers voor de andere rekenkern.

Om ten volle gebruik te kunnen maken van deze meerdere rekenkernen moeten software-ontwikkelaars ook hun programma's aanpassen. Immers waar het vroeger voldoende was één draad te hebben per programma, is het nu interessanter om meerdere draden per programma te hebben zodanig dat de draden parallel kunnen uitvoeren op de verschillende rekenkernen. Meestal kunnen deze verschillende draden echter niet geheel onafhankelijk van elkaar uitvoeren omdat ze gegevens moeten delen of wachten op elkaar. Daarom voorziet software in synchronisatiemethodes. Deze synchronisatie is nodig om een correcte

uitvoering van het programma te verzekeren, maar heeft als nadeel dat draden opnieuw een invloed hebben op elkaars uitvoering, net zoals bij de gedeelde hardware componenten.

Deze twee types van interacties tussen draden zijn bepalend voor de prestatie van meerdradige programma's, maar ze maken het tevens ook complex om deze prestatie te analyseren. Daarom introduceren we in dit proefschrift drie nieuwe methodes, genaamd *criticality stacks, bottle graphs* en *speedup stacks*, die het eenvoudig maken om de prestatie te analyseren van meerdradige programma's. De methodes tonen op een visuele manier aan software en hardware ontwikkelaars wat er gebeurt tijdens de uitvoering van een parallel programma.

Onze eerste methode, *criticality stacks*, is bruikbaar voor het analyseren van onevenwicht tussen draden. Om deze stacks te construeren stellen we een nieuwe criticaliteitsmetriek voor, die onafhankelijk is van het type synchronisatie. De metriek wordt berekend aan de hand van de tijd dat een draad actief is, en het aantal andere draden dat tegelijk uitvoert. Gebruik makend van deze metriek splitsen criticality stacks de uitvoeringstijd van een applicatie op in een deel voor iedere draad. Hoe groter dit deel is voor een draad, hoe kritischer deze draad is voor de applicatie. Om deze metriek te berekenen tijdens de uitvoering van een programma stellen we een kleine hardwarecomponent voor. Deze component bevindt zich niet op het kritisch pad van de processor en verbruikt een kleine hoeveelheid aan extra vermogen. In dit werk gebruiken we criticality stacks voor het analyseren van onevenwicht tussen draden voor een verzameling van benchmarks, het sturen van optimalisatie van software, het dynamisch optimaliseren van de prestatie en het reduceren van het energieverbruik van parallelle applicaties.

De tweede methode, *bottle graphs*, stelt iedere draad van een meerdradig programma voor als een rechthoek in een grafiek. De hoogte van de rechthoek wordt berekend door middel van onze criticaliteitsmetriek, de breedte stelt het parallellisme van een draad voor (hoeveel andere draden er tegelijk met die draad actief zijn). De oppervlakte van de rechthoek is gelijk aan de totale uitvoeringstijd van de draad. De rechthoeken worden dan op elkaar gestapeld in een grafiek en gesorteerd naar gelang hun breedte, met de smalste rechthoeken bovenaan. Dit betekent dat draden met een beperkt parallellisme bovenaan in de grafiek zitten, als het ware in de hals van de fles waardoor we ze beschouwen als "bottlenecks" voor de applicatie. Hoewel bottle graphs geconstrueerd kunnen worden met de hardwarecomponent die we ontwierpen voor criticality stacks, hebben we een tweede profileringstechniek uitgewerkt volledig in software. Hierdoor kunnen we bottle graphs construeren van ongewijzigde programma's die uitvoeren op hedendaagse processors (en niet op een simulator). In dit werk gebruiken we bottle graphs voor het analyseren van de prestatie van Java programma's uitvoerend op Jikes RVM. We analyseren de prestatie van zowel applicatiedraden (die afkomstig zijn van de programma's), als servicedraden (die afkomstig zijn van de virtuele machine).

Onze derde methode, *speedup stacks*, toont de bereikte speedup (versnelling t.o.v. sequentiële uitvoering) van een applicatie en de verschillende componenten die speedup beperken in een gestapelde grafiek. De totale hoogte van de stapel is de maximaal bereikbare speedup; de onderste component in de stapel toont de bereikte speedup. De andere elementen in de stapel tonen de componenten die de speedup beperken en hun relatieve impact op de speedup. De intuïtie achter het concept van een speedup stack is dat door het reduceren van de invloed van een bepaalde component, de speedup van een applicatie proportioneel toeneemt met de grootte van die component in de stapel. In dit werk stellen we twee versies van speedup stacks voor. Onze eerste versie gebruikt extra hardwareondersteuning om speedup stacks te construeren tijdens de uitvoering van een programma. In deze versie bevat de speedup stack de volgende componenten: interferentie in de gedeelde cache en de geheugenhiërarchie, actief spinning, tijd uitgescheduled, en onevenwicht tussen draden. De profileringsmethode gebruikt een speciaal ontwikkelde tellerarchitectuur voor het opmeten van positieve en negatieve interferentie op de prestatie van draden. De extra overhead van deze profileringsmethode is beperkt tot 1,1 KB per rekenkern, dus ongeveer 18 KB voor een processor met 16 rekenkernen. In dit proefschrift gebruiken we deze versie van speedup stacks voor het identificeren van schalingsproblemen, het classificeren van benchmarks gebaseerd op hun schalingsgedrag, en voor het begrijpen van de prestatie van een gedeelde cache. Onze tweede versie van speedup stacks richt zich op Java-programma's. In deze versie bestaat de stapel uit andere componenten, namelijk garbage collection, sequentiële delen, onevenwicht tussen draden, synchronisatie en interferentie in de hardware. Voor het opmeten van deze speedup stacks hergebruiken we de profileringsmethode die we ontwierpen voor het genereren van bottle graphs, maar we passen deze methode aan om onze speedup stacks te kunnen construeren. Deze tweede versie van speedup stacks gebruiken we in dit werk voor het bestuderen van het schalingsgedrag van Java-applicaties die uitvoeren op Jikes RVM.

Als besluit geloven we dat deze drie methodes inzichtelijk zijn voor zowel software- als hardware-ontwikkelaars voor het begrijpen van het schalingsgedrag, het identificeren van bottlenecks, en het optimaliseren van de prestatie van meerdradige programma's. Dit is geen eenvoudige taak omdat er interacties tussen draden zijn zowel in de hardware, omwille van gedeelde componenten, als in de software door synchronisatie tussen draden.

# Summary

For many years the increase in transistor density coming from Moore's law combined with Dennard's scaling made it possible for chip manufacturers to increase single-core performance. During this period the trend of decreased transistor size resulted in increased clock frequency and pipeline width of the processor, as exemplified by the Intel Pentium 4 design. However, because of power and cooling constraints related to it, this trend came to an end and chip manufacturers went in the direction of multi-core processors, where instead of one core doing all the work, there are multiple cores available on the same chip.

While these multi-core processors contain several cores that are able to work independently from each other, the different cores also share resources on the chip, for example, a last-level cache, an on-chip interconnection network, or a memory bus. Sharing these resources increases the utilization of the components, but comes with a drawback that cores can affect each other's performance. For example, in case of a shared cache, one core can evict data of another core leading to additional misses for the latter.

Also, in order to fully benefit from the available cores, software writers have to change the way they design their programs. Instead of having just one thread per program, it is now more beneficial to have multiple threads in the same application. Typically, those multiple threads can not work completely independently from each other, because at some point in their execution they have to share some data or they have to wait for each other. Therefore, software provides methods for synchronization. While this synchronization is necessary to achieve a correct execution of the program, it also has the disadvantage that – just like resource sharing – threads can affect each other's performance.

These two types of interactions between threads are determinative to the performance of a multi-threaded application, but make it also difficult to analyze the performance of these applications. Therefore we introduce three new methods in this dissertation, called *criticality stacks, bottle graphs* and *speedup stacks*, to facilitate multi-threaded application performance analysis. Our methods visually show programmers and hardware designers what is going on during the execution of parallel programs.

The first method, *criticality stacks*, is useful for visual analysis of parallel imbalance between threads. They are constructed using a novel, intuitive criticality metric that is independent of synchronization primitives, and takes into account both a thread's active running time and the number of co-executing threads. Using this metric, criticality stacks break down the total execution time of an application based on each thread's criticality. The higher the share of a thread in the stack, the more critical the thread is, meaning that the thread is more determinative of execution time. For calculating a thread's criticality value online during the execution of an application, we present a small hardware component. This component is off the processor's critical path and consumes a very small amount of power. We use criticality stacks in this work for analyzing parallel imbalance in a set of applications, guiding software source code optimization, dynamically optimizing the performance, and reducing the energy consumption of parallel programs.

The second method, *bottle graphs*, shows each thread of a multi-threaded application as a box in a stacked bar graph. The height of a box is a thread's execution time share (using our criticality metric) and the width is its parallelism (which is the average number of co-running threads). As a result, the total area of a box is equal to a thread's total running time. We sort boxes according to their width (parallelism) with the narrowest ones at the top, meaning that the threads in the neck of the bottle have low parallelism and therefore reveal themselves as parallel bottlenecks of the application. While we can construct bottle graphs with the hardware component that we designed for constructing criticality stacks, we designed a second profiling tool that is implemented completely in software. This allows us to construct bottle graphs for unmodified applications running on native hardware with minimal overhead. In this work we use bottle graphs to analyze Java applications running on top of Jikes RVM. We do an analysis of the behavior of the application threads (coming from the benchmarks), as well as the additional service threads introduced by the runtime, such as for memory management.

Our third method, *speedup stacks*, visualizes the achieved speedup of an application and the various scaling delimiters as a stacked bar.

The total height of the stack is equal to the maximum achievable speedup over its single-threaded execution (excluding superlinear scaling), the bottom component in the stack shows the actual speedup, whereas the components on top of it show the various scaling delimiters and their relative impact on speedup. The intuition behind it is that by reducing the impact of a speedup delimiter, speedup improves proportional to the height of the component in the stack. In this disseratation we propose two versions of speedup stacks. The first version uses additional hardware support for computing a speedup stack during the execution of a program, at low overhead. The various speedup delimiters we include in these speedup stacks are: LLC and memory subsystem interference, spinning, yielding, imbalance. The profiling tool uses a dedicated counter architecture to estimate the impact of both negative and positive interference on the performance on threads. Hardware overhead of this tool is limited to 1.1 KB per core, or a total of 18 KB for a 16-core CMP. We use this version of speedup stacks in this work for identifying scaling bottlenecks, classifying benchmarks based on their scaling delimiters, and for understanding LLC performance. The second version of speedup stacks targets managed language applications, like Java programs. For this version of speedup stacks we include a different set of scaling delimiters: garbage collection, sequential parts, thread imbalance, synchronization between threads, and hardware interference. For constructing these speedup stacks, we extend the profiling tool that we designed for generating bottle graphs. This profiling tool is completely implemented in software. We use these speedup stacks for analyzing scaling behavior of Java applications running on Jikes RVM.

We believe the three methods proposed in this dissertation are fundamental to both software and hardware designers for understanding the behavior, identifying scaling bottlenecks, and optimizing the performance of multi-threaded applications. This analysis is not trivial due to threads that interact with each other in software through synchronization in the code and work balance, and in hardware through sharing resources.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| ATD | Auxiliary Tag Directory |
| CMP | Chip-Multiprocessor |
| CPI | Cycles Per Instructions |
| DTLB | Data Translation Lookaside Buffer |
| GC | Garbage Collection |
| HPC | High-Performance Computing |
| IPC | Instructions Per Cycle |
| JVM | Java Virtual Machine |
| L1 | Level 1 |
| L2 | Level 2 |
| L3 | Level 3 |
| LLC | Last-Level Cache |
| MLP | Memory-Level Parallelism |
| MPI | Message Passing Interface |
| MSHR | Miss Status Holding Register |
| OS | Operating System |
| PARSEC | Princeton Application Repository for Shared-Memory Computers |
| RVM | Research Virtual Machine |

| | |
|---|---|
| SMP | Shared-Memory Processor |
| SMT | Simultaneous Multi-Threading |
| SPLASH | Stanford Parallel Applications for Shared-Memory |
| WAIT | Whole-system Analysis of Idle Time |

# Chapter 1

# Introduction

## 1.1 Motivation

While multi-core processors improve overall chip throughput and utilization, sharing hardware resources among the cores in caches, the on-chip interconnection network, and memory bus, leads to an unpredictable performance of the individual threads running on a multi-core processor. This is because co-executing threads interfere with each other in the shared hardware resources. This interference between threads can either have a negative impact on the performance of an application, meaning that the execution time of the threads becomes longer during multi-core execution compared to an isolated execution of the threads (which is without competing for hardware resources), or it can have a positive impact on performance, meaning that the execution time of threads becomes shorter during multi-core execution.

The advent of multi-core processors also poses new challenges to software designers, because they have to find ways to make use of those multiple available cores. These days, most programmers do this by parallelizing their software, which means that, instead of having one thread doing all the work, the work is now divided between several threads that are each doing a part of the total work. While this idea is simple, the implementation of this concept is often complicated, because the different threads typically have to share data, which means there has to be synchronization between the threads.

Just like resource sharing, this synchronization between threads leads to threads that affect each other's performance, resulting in a poor scaling of an application. This means that the execution time of a multi-threaded application is not proportional to the number of threads or available cores in the system. Therefore, it is vital for both programmers and processor designers to understand multi-threaded application behavior in order to optimize performance and to design future hardware.

## 1.2 Key Challenges

Analyzing multi-threaded programs and identifying scaling bottlenecks is very challenging, but it is necessary to obtain good parallel performance. The reasons why it is a complicated task, is that threads interact with each other because of resource sharing in the underlying hardware and due to synchronization between the threads. This means that, if we want to analyze the performance of this type of applications, we have to measure and quantify the impact of these interactions across various layers in the computer system stack, including both hardware and software. Apart from measuring, it is also challenging to represent this data into graphs that can be easily interpreted by software developers without having to know the exact details about the underlying hardware platform.

Besides analyzing, it's also challenging to optimize the performance of multi-threaded applications. We will see later in this work that accelerating a particular thread of an application at a certain point in the execution leads to a performance gain, while accelerating other threads does not change performance, but wastes energy. The challenge here is to identify which threads need to be accelerated at which moment during the execution.

## 1.3 Contributions in This Dissertation

This dissertation presents new methods for analyzing and optimizing the performance of multi-threaded applications running on modern multi-core processors.

### Contribution #1: Per-Thread Cycle Accounting in Multi-Core Processors

This work proposes a hardware-efficient per-thread cycle accounting architecture for multi-core processors. The counter architecture tracks per-thread progress in a multi-core processor, detects how inter-thread interference affects per-thread performance, and predicts the execution time for each thread if run in isolation. The counter architecture captures the effect of interference misses due to cache sharing as well as increased memory access latency due to resource and memory bandwidth sharing in the memory subsystem. The proposed method accounts for 74.3% of the interference cycles, and estimates per-thread progress within 14.2% on average across a large set of multi-program workloads. Hardware cost is limited to 7.44 KB for an 8-core processor — a reduction by almost 10× compared to prior work while being 63.8% more accurate. Chapter 6 details on the implementation of this counter architecture. Making system software progress-aware improves fairness by 22.5% on average over progress-agnostic scheduling, this will be discussed in Chapter 7.

This work has been published in ACM Transactions on Architecture and Code Optimization (TACO) and has been presented at the 2013 International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC):

> K. Du Bois, S. Eyerman, and L. Eeckhout. Per-thread Cycle Accounting in Multicore Processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–22, Jan. 2013

### Contribution #2: Speedup Stacks

Multi-threaded workloads typically show sublinear speedup on multi-core hardware, i.e., the achieved speedup is not proportional to the number of cores and threads. Sublinear scaling may have multiple causes, such as poorly scalable synchronization leading to spinning and/or yielding, and interference in shared resources such as the last-level cache (abbreviated as LLC) as well as the main memory subsystem.

In this work, we propose the speedup stack, which quantifies the impact of the various scaling delimiters on multi-threaded application speedup in a single stack. We describe a mechanism for computing speedup stacks on a multi-core processor, and we find speedup stacks to be accurate within 5.1% on average for sixteen-threaded applications. We present several use cases: we discuss how speedup stacks can be used to identify scaling bottlenecks, classify benchmarks, optimize performance, and understand LLC performance. We introduce speedup stacks in Chapter 3 and discuss them further in Chapter 6.

Speedup stacks have been presented at the 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS):

> S. Eyerman, K. Du Bois, and L. Eeckhout. Speedup Stacks: Identifying Scaling Bottlenecks in Multi-Threaded Applications. In *Proceedings of the International Symposium on Performance Analysis of Software and Systems (ISPASS)*, pages 145–155, Apr. 2012

### Contribution #3: Criticality Stacks

Due to synchronization, certain threads make others wait, because they hold a lock or have yet to reach a barrier. We call these critical threads, i.e., threads whose performance is determinative of program performance as a whole. Identifying these threads can reveal numerous optimization opportunities, for the software developer and for hardware.

In this work, we propose a new metric for assessing thread criticality, which combines both how much time a thread is performing useful work and how many co-running threads are waiting. We show how thread criticality can be calculated online with modest hardware additions and at low overhead. We use our metric to create criticality stacks that break down total execution time into each thread's criticality component, allowing for easy visual analysis of parallel imbalance. We introduce criticality stacks in Chapter 3.

To validate our criticality metric, and demonstrate it is better than previous metrics, we scale up the frequency of the most critical thread and show it achieves the largest performance improvement. We then demonstrate the broad applicability of criticality stacks by using them to perform three types of optimizations: (1) program analysis to remove parallel bottlenecks, (2) dynamically identifying the most critical thread and accelerating it using frequency scaling to improve performance, and (3) showing that accelerating only the most critical thread allows for targeted energy reduction. We discuss this in Chapter 4 and 7.

This work has been presented at the 2013 International Symposium on Computer Architecture (ISCA):

> K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout. Criticality Stacks: Identifying Critical Threads in Parallel Programs Using Synchronization Behavior. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 511–522, June 2013

### Contribution #4: Bottle Graphs

In our fourth contribution, we present bottle graphs, a powerful analysis tool that visualizes multi-threaded program performance, in regards to both per-thread parallelism and execution time. Each thread is represented as a box, with its height equal to the share of that thread in the total program execution time, its width equal to its parallelism, and its area equal to to the thread's total execution time. The boxes of all threads are stacked upon each other, leading to a stack with height equal to the total program execution time. Bottle graphs show exactly how scalable each thread is, and thus guide optimization towards those threads that have a smaller parallel component (narrower), and a larger share of the total execution time (taller), i.e., towards the 'neck' of the bottle.

Using light-weight OS modules, we calculate bottle graphs for unmodified multi-threaded programs running on real processors with an average overhead of 0.68%. To demonstrate their utility, we do an extensive analysis of 12 Java benchmarks running on top of Jikes RVM, which introduces many virtual machine service threads. We not only reveal and explain scalability limitations of several well-known Java benchmarks; we also analyze the reasons why the garbage collector itself does not scale, and in fact performs optimally with two collector threads for all benchmarks, regardless of the number of application threads. Finally, we compare the scalability of Jikes versus the OpenJDK JVM. We demonstrate how useful and intuitive bottle graphs are as a tool to analyze scalability and help optimize multi-threaded applications. We introduce bottle graphs in Chapter 3 and discuss them further in Chapter 5.

Bottle graphs have been presented at the 2013 ACM SIGPLAN International Conference on Object Oriented Programming, Systems Languages and Applications (OOPSLA):

K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-threaded Applications. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming, Systems Languages and Applications (OOPSLA)*, pages 355–372, Oct. 2013

### Contribution #5: Extension on Bottle Graphs and Speedup Stacks

In our fifth contribution, we provide an extension to our previous bottle graphs and speedup stacks papers. We use bottle graphs for analyzing per-thread performance of Java applications running on Jikes RVM. Because it is hard to analyze how the applications themselves scale using bottle graphs, we extend our previously proposed speedup stacks to make them suitable for managed language programs like Java applications. We include the following scaling delimiters to more accurately analyze the scalability of the service threads of managed language programs: garbage collection, sequential parts of the application, thread imbalance, synchronization between threads and hardware interference. For constructing these new speedup stacks, we extend our previously proposed light-weight OS modules that we used for generating bottle graphs. We thus generate speedup stacks for unmodified Java applications running on native hardware at very low overhead. The speedup stacks lead to a better understanding of the causes and contributions of limited speedup of multi-threaded Java programs. We discuss these speedup stacks in Chapter 3 and 6.

This work is submitted to ACM Transactions on Programming Languages and Systems (TOPLAS):

K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout. Analyzing Scaling Behavior of Managed Runtime Applications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2014. Under review

### How the Contributions Tie Together

Table 1.1 gives an overview of the different contributions in this dissertation. The first contribution, the per-thread cycle accounting architecture, uses a dedicated hardware component for measuring the impact of interference between threads, due to hardware resource sharing.

| | *Per-Thread Cycle Accounting* | *Speedup Stacks* | *Criticality Stacks* | *Bottle Graphs* |
|---|---|---|---|---|
| Implemented with: | | | | |
| - hardware | contr. #1 | contr. #2 | contr. #3 | — |
| - software support | — | contr. #5 | — | contr. #4 |
| Used for: | | | | |
| - native | contr. #1 | contr. #2 | contr. #3 | — |
| - managed language applications | — | contr. #5 | — | contr. #4 |
| Visualization tool | ✘ | ✔ | ✔ | ✔ |

**Table 1.1:** Overview of contributions in this dissertation.

We evaluate our counter architecture on multi-programmed workload mixes, consisting of SPEC CPU2006 benchmarks.

In this dissertation we have two versions of speedup stacks. The first version, found in contribution #2, uses an extended version of the cycle accounting architecture from the first contribution for generating speedup stacks. We use this version of speedup stacks for evaluating the performance of applications from the PARSEC, SPLASH-2 and Rodinia benchmark suites. Our second version of speedup stacks, from contribution #5, is constructed using a software implementation. For this second version, we extend our original speedup stacks to be able to analyze Java applications. We use these speedup stacks for analyzing the performance of multi-threaded Java benchmarks that come from the DaCapo suite, running on Jikes RVM.

In contribution #3, we propose criticality stacks, and a corresponding profiling tool that uses hardware support to generate those stacks. We consider a set of benchmarks from the PARSEC, SPLASH-2 and Rodinia suites for our study with criticality stacks. Contribution #4 presents bottle graphs, which we generate using a profiling tool implemented in software. We use bottle graphs for performance analysis of Java applications (both single- and multi-threaded applications) running on Jikes RVM.

Speedup stacks, criticality stacks and bottle graphs all provide a visual representation that facilitates an intuitive analysis of the performance of multi-threaded applications.

## 1.4 Other Research Activities

Besides the contributions mentioned above, we also performed research on evaluating the energy efficiency of computer systems. These results are not discussed in this dissertation, but we refer the interested reader to the respective publication.

### Evaluating Computer System Energy Efficiency

Energy efficiency is a key design concern in contemporary processor and system design, in the embedded domain as well as in the enterprise domain. The focus on energy efficiency has led to a number of power benchmarking methods recently. For example, EEMBC released EnergyBench, and SPEC released SPECpower to quantify a system's energy efficiency; also academics have proposed power benchmarks, such as JouleSort. A major limitation for each of these proposals is that they are tied to a specific benchmark, and hence, they provide limited insight with respect to why one system is more energy-efficient than another.

In this contribution we propose SWEEP, Synthetic Workloads for Energy Efficiency and Performance evaluation, a framework for generating synthetic workloads with specific behavioral characteristics. We employ SWEEP to generate a wide range of synthetic workloads while varying the instruction mix, ILP, memory access patterns, and I/O-intensiveness; and we use SWEEP to evaluate the energy efficiency of commercial computer systems across the workload space and learn about how the energy efficiency of a computer system is tied to workload characteristics.

This work also presents the Energy-Delay Diagram (EDD), a novel method for visualizing energy efficiency. The EDD clearly illustrates the energy versus performance trade-off, and provides more intuitive insight than the traditionally used EDP and ED$^2$P metrics.

We refer the interested reader to the following paper that was presented at the 2011 International Conference on High-Performance and

Embedded Architectures and Compilers (HiPEAC):

> K. Du Bois, T. Schaeps, S. Polfliet, F. Ryckbosch, and L. Eeckhout. SWEEP: Evaluating Computer System Energy Efficiency Using Synthetic Workloads. In *Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, pages 159–166, Jan. 2011

## 1.5 Overview of This Dissertation

This dissertation is organized as follows. In Chapter 2, we provide background on resource sharing between threads in a multi-core processor and discuss synchronization in multi-threaded applications. In Chapter 3, we introduce our three new methods for analyzing the performance and identifying scaling bottlenecks of multi-threaded applications. We further discuss criticality stacks, bottle graphs and speedup stacks in Chapter 4, 5, and 6 respectively. These chapters explain how the new methods are constructed, the experimental setup we used, and results about performance analysis of applications. We show how application performance can be optimized in Chapter 7. Finally we present our conclusions and talk about future work in Chapter 8.

# Chapter 2

# Background

*In this chapter we provide a background about contemporary multi-core processors, how threads interfere with each other in the shared hardware resources, and discuss synchronization in multi-threaded applications.*

## 2.1 Multi-Core Processors

Moore's law says that the number of transistors on a single chip doubles every two years [48]. As a result of this increase in transistor density, chip manufacturers today are able to put multiple cores on a single chip. Those designs are called multi-core processors or chip-multiprocessors (CMPs) [50]. Examples of contemporary multi-core processors are the Intel Core i7, AMD Opteron, IBM POWER8, etc. These general-purpose processors employ a limited number of cores, typically in the range of 4 to 8 cores, but given the continuous transistor density improvements, it is expected that this number will increase in the coming years, as exemplified by Intel's Xeon Phi with 62 cores on a single chip.

Figure 2.1 shows a high-level overview of a modern multi-core processor, together with an off-chip memory component. In this example the processor contains 4 cores. The cores are connected to their private L1 caches, which are typically separate for instructions and data. Besides a private L1, each core also has a private L2 cache in this example. The cores are connected to a shared last-level cache, which is the L3 cache, by making use of an interconnection network. In case a core wants to access data that cannot be found in one of the caches, the core

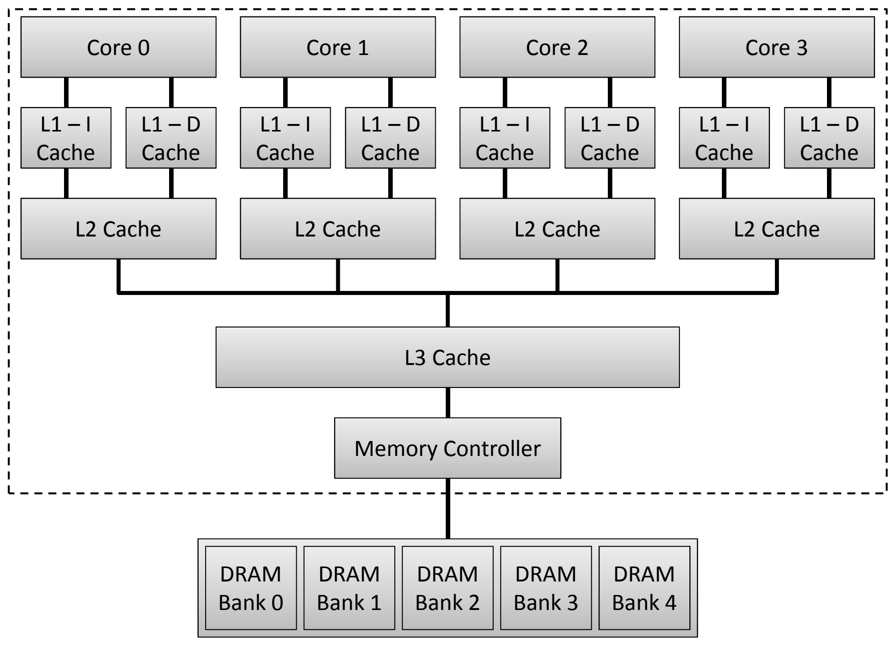


**Figure 2.1:** Example of a multi-core processor along with an off-chip memory component. The dashed line indicates the chip boundary.

sends a request to the memory controller, which fetches the data from main memory. Main memory is normally placed off-chip and is connected to the chip using a memory bus. Main memory itself exists of several memory banks, which can handle memory accesses in parallel, thereby enabling memory-level parallelism (MLP).

### 2.1.1 Resource Sharing

As illustrated in Figure 2.1, multi-core processors share resources among the cores, such as caches, on-chip interconnection network, memory controllers, off-chip bandwidth, memory banks, etc. Resource sharing increases hardware utilization, adds flexibility for a processor to adapt to varying workload demands (e.g., a thread with a large working set can allocate a large fraction of the shared cache), and can improve performance (e.g., fast communication between cores through the on-chip interconnection network and shared cache).

However, resource sharing also comes with a significant drawback: co-executing hardware threads may affect each other's performance.

For example, a thread allocating a large fraction of the shared cache may introduce additional interference misses for other threads [45, 64]; likewise, memory accesses by a thread may close open pages in memory, thereby increasing memory access time for other threads.

These inter-thread interferences may or may not have an effect on per-thread performance depending on whether memory accesses can be hidden by doing other useful work. As a result, hardware resource sharing may affect the performance of a multi-threaded application in unpredictable ways and may possibly lead to undesirable properties such as an unbalanced performance across co-executing threads of a parallel workload, thread starvation, etc. Besides that, the impact of resource sharing on application performance is proportional to the number of co-running threads in the application. As a result, the parallel speedup of applications executing on multi-core processors does not scale linearly in the number of threads, but is limited by the impact of resource sharing on per-thread performance.

Therefore, when analyzing the scaling behavior of multi-threaded applications, it is vital to accurately quantify this impact on the performance of an application.

### 2.1.2 Sources of Thread Interference

Co-executing threads on a multi-core processor interfere with each other in each of the shared resources, which leads to different interference effects. We will now discuss how this interference affects the performance of threads. For now, we assume a single thread per core, hence we use the terms 'thread' and 'core' interchangeably.

#### Last-level cache

Sharing the last-level cache (the L3 cache in Figure 2.1) between threads leads to extra interference misses due to threads evicting each other's data. We refer to these interference misses as *inter-thread misses*. In contrast, we define *intra-thread misses* as misses that also occur during isolated execution, i.e., when the thread runs alone on the processor. Inter-thread misses do not occur during isolated execution and hence, their performance impact is potentially detrimental to per-thread performance: these memory references would be serviced by the LLC in isolated execution but turn into long-latency memory accesses during multi-core execution.

In case of a multi-threaded application, sharing the last-level cache can also have a positive impact on performance. This happens when a thread loads data into the cache that later can be used by other threads. We define an *inter-thread hit* as a hit that would be a miss during isolated execution but becomes a hit during multi-core execution.

### Interconnection network

The on-chip interconnection network connects the cores to the shared cache (and to each other). A request of one core can be delayed due to a request by another core. Conflicts in the interconnection network thus prolong both the LLC hit and miss latency compared to isolated execution. Prolonging the LLC hit time due to conflicts in the interconnection network is unlikely to significantly affect per-thread performance, because the LLC hit latency (even with the additional conflict latency) is small enough so that it is effectively hidden on superscalar processor cores through out-of-order execution in a balanced design [25]. For LLC misses, the additional conflict latency may have a significant effect (i.e., the additional penalty can not be hidden) because the processor cannot make progress while handling the LLC miss because of its long latency.

### Memory bus

As with the interconnection network, a memory request issued by a core can hold the bus between the LLC and main memory, possibly delaying requests by other cores. This causes memory accesses to take longer, which may have a significant impact on performance.

### Memory bank effects

While main memory typically consists of a number of memory banks that can handle memory accesses in parallel, each bank can handle only one access at a time. This implies that while a bank is busy processing an access of a core, no other requests to that bank from other cores can be serviced. This increases the memory access time for the other cores.

An additional effect occurs in case of an open-page policy. Consider an example in which a thread accesses the same page twice and there are no intervening memory accesses to another page in the same bank, i.e., the page is loaded in the row buffer and both accesses are serviced from the row buffer. Now, another thread may interfere and may initiate a memory access to that same bank (but a different page) between the two memory requests by the first thread. This memory access will cause the row buffer to be written back to the memory bank, and a new page to be loaded in the row buffer. The second memory access by the first thread will now see a row miss (instead of a row hit) and will need to load the page again into the row buffer. In other words, this second memory access will see a longer latency during multi-core execution than it would see during isolated execution.

#### Prefetcher

A hardware prefetcher tries to fetch data from a higher memory level into a lower memory level before a core sends a request for accessing the data. This way, hardware tries to anticipate future accesses to both instructions and data, and thereby improve the hit rate of a memory level. Typically there is a prefetcher between the last-level cache and main memory.

Prefetch requests in a memory system usually have a lower priority than read or write requests. Therefore, interference in this component happens when prefetch requests that would be timely in isolated execution are delayed during multi-core execution because of requests from the other cores, and as a result now become a miss for the core. Secondly, despite the lower priority of prefetch requests, they may still congest the memory subsystem because they occupy the memory bus, pollute caches, etc.

### 2.1.3 Quantifying Thread Interference

In this section we show that interference between threads in shared resources has a significant impact on performance. We do this by running experiments using the gem5 simulator [6], and simulating processors with 1, 2, 4 and 8 cores running multiple single-threaded SPEC CPU2006 workloads (more information about our simulated configurations can be found in Table 6.2 on page 107). We then quantify the impact of this interference on per-thread performance, and we identify the contribution of different sources of interference.

We define *interference* as the relative increase in execution time be-

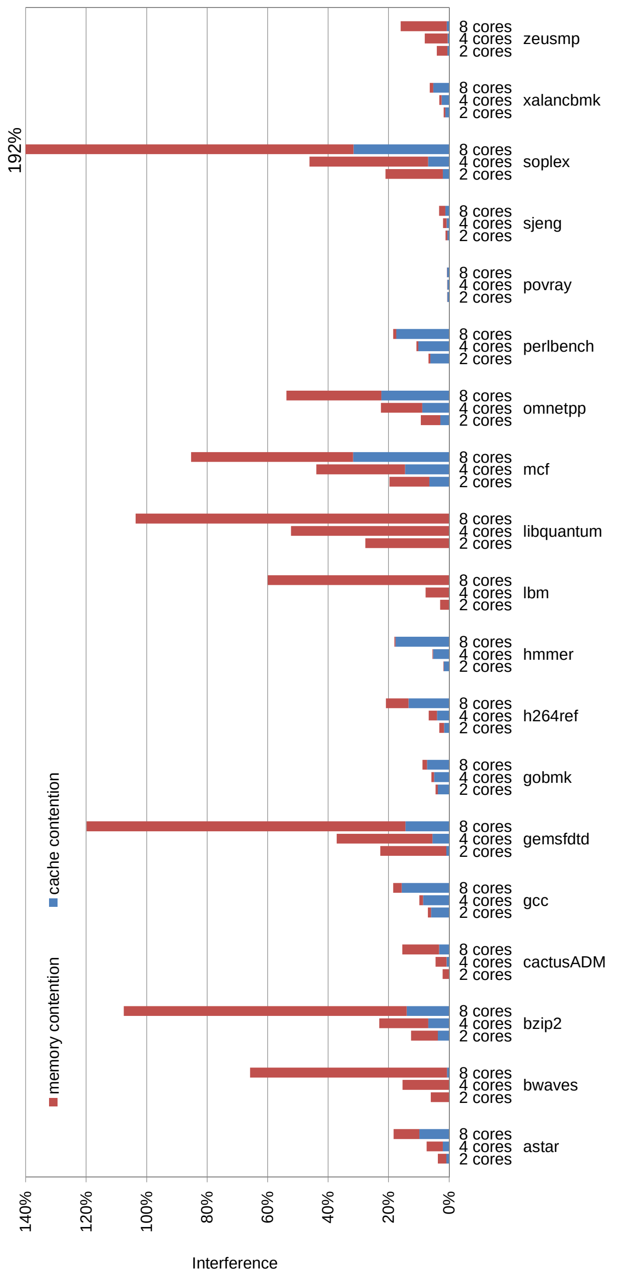


**Figure 2.2:** Impact of inter-thread interference on per-thread performance for 2, 4 and 8 cores, breaking up interference in cache versus memory contention (average interference is reported across a set of job mixes per benchmark and assuming hardware prefetching).

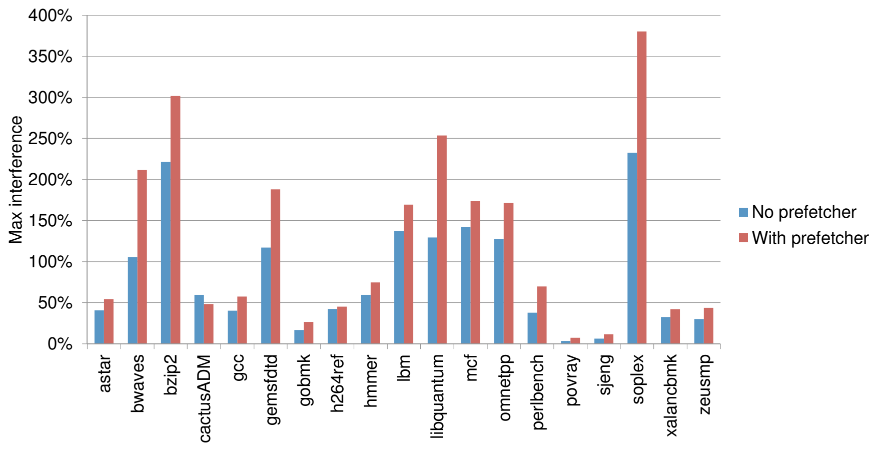


**Figure 2.3:** The impact of prefetching on interference on an eight-core system (maximum interference is reported across 10 job mixes per benchmark).

tween multi-core and isolated execution:

$$Interference = \frac{T_{multi\text{-}core} - T_{isolated}}{T_{isolated}}. \tag{2.1}$$

*Interference* thus quantifies the increase in execution time on a multi-core processor due to interference relative to isolated single-core execution. Through detailed simulation we find that interference is significant, and that it increases with the number of cores: 9.3% on average for 2 cores, 19.5% for 4 cores, and 55.4% for 8 cores. The reason why interference increases with core count is that an increasing number of cores put increasingly more pressure on the shared resources, and hence, per-thread performance is affected more significantly.

To understand the relative contributions of the different sources of interference, Figure 2.2 makes a distinction between the interference due to inter-thread misses in the shared cache versus resource and bandwidth sharing in the memory subsystem (memory bus, memory banks and open row policy). Some benchmarks seem to suffer more from cache sharing, whereas other benchmarks suffer more from sharing the memory subsystem. These interference numbers illustrate that the shared resources have substantial impact on per-thread performance, and by consequence, estimating interference is non-trivial (i.e., the null predictor would be highly inaccurate).

Figure 2.3 quantifies the impact of hardware prefetching on interference. We model a stride prefetching scheme that prefetches the next four cache blocks if a stride is detected. Now the maximum interference level observed increases from 2.3× without prefetching to up to 3.8× with prefetching. The reason is that hardware prefetching puts even more pressure on the memory system's shared resources, which in its turn affects per-thread progress. In particular, a core that issues many prefetch requests may congest the memory subsystem and thereby degrade other cores' performance.

## 2.2 Multi-Threaded Applications

In order to take advantage of multi-core processors, software has to provide enough parallel work to make use of the available resources in order to continue the trend of ever-improving performance. Multi-threaded programs that try to use these resources, inherently introduce synchronization to ensure correct execution, for example because they share data among the threads.

Figure 2.4 shows three common examples of synchronization. In Figure 2.4(a) we show a barrier. A barrier is a synchronization primitive that imposes ordering and denotes a point in the execution beyond which a thread is only allowed to go after all other threads have reached that point. The result of a barrier is that the execution of a thread is halted until all threads have reached the barrier. A barrier can be shared across all threads, or between a subset of threads.

Figure 2.4(b) illustrates the use of critical sections. Critical sections are typically implemented using locks to guarantee atomicity when modifying shared data. Critical sections do not impose a particular ordering of execution, but they prevent threads from reading and modifying the same data concurrently. An alternative to using locks for guaranteeing atomicity in critical sections is transactional memory.

Figure 2.4(c) shows the use of producer-consumer synchronization. In this case threads can only proceed with their calculation after the needed data is produced by other threads. (In the example T2 and T3 have to wait until T0 and T1 finished their calculation.)

When threads are waiting due to synchronization, they can either be in a spinning or yielding state. Spinning means a thread is continuously checking the state of a synchronization variable inside a loop, which is very compute-intensive. Therefore, there is a second state,

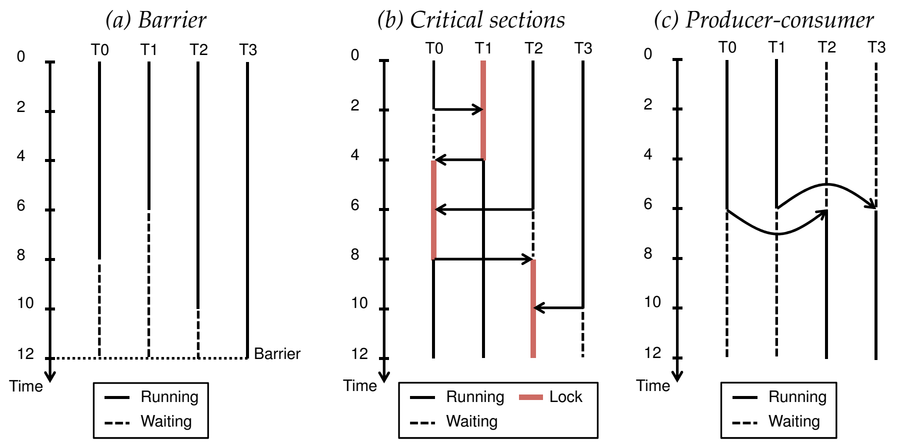


**Figure 2.4:** Examples of synchronization between threads of a multi-threaded application.

called yielding in this work, which means that the operating system schedules out the thread. In this state the thread does no longer occupy the core. However, this approach has the disadvantage of having a larger performance penalty compared to spinning (because the operating system has to schedule the threads in and out).

While synchronization is necessary, it also results in threads waiting for each other, limiting performance and scalability, and wasting energy. This means that threads in a multi-threaded application have an impact on each other's performance, just like they do due to resource sharing in the underlying hardware of a multi-core processor.

## 2.3 Summary

In this chapter, we identified two reasons why threads have an impact on each other's execution, and consequently limit parallel speedup of a multi-threaded application. First, they share resources in the hardware of a multi-core processor. This leads to interference among co-executing threads because of contention effects in the shared resources, such as caches, off-chip bandwidth, memory banks, etc. We quantified that this interference is significant and increases with the number of cores.

Secondly, while synchronization is necessary for achieving correct execution of a multi-threaded application, it also causes threads to wait for each other. This waiting of threads results in certain threads making faster progress than others, leading to an imbalance between the different threads, and some threads that are more critical to performance than others.

We conclude this chapter by saying that apart from the level of the hardware, threads also affect each other's performance on the level of the application. This implies that if we want to analyze the performance and scalability of a multi-threaded application, we have to take into account both levels of interference (both in hardware and in software).

# Chapter 3

# Performance Analysis Methods

*In this chapter we introduce three new performance analysis methods for evaluating the performance and scalability of multi-threaded applications.*

## 3.1 Introduction

Analyzing the performance and scalability of a multi-threaded application is not trivial. As we discussed in the previous chapter, threads interact with each other due to synchronization and resource sharing in the hardware. These complicated interactions make it difficult to analyze performance. However, one of the key needs to efficient parallel programming is to have the appropriate tools to analyze parallel performance. In particular, a software developer needs analysis tools to identify the performance scaling bottlenecks, not only on current hardware but also on future hardware with many more cores than are available today; likewise, computer architects need analysis tools to understand the behavioral characteristics of workloads to design and optimize future hardware.

Therefore, we propose three new methods for analyzing parallel program performance in this chapter. The first method, called *criticality stacks*, is useful for understanding parallel (im)balance between threads of a multi-threaded application. The second method, *bottle graphs*, visualizes parallel performance bottlenecks by quantifying both execution time and parallelism for each thread. Finally, we present *speedup stacks*,

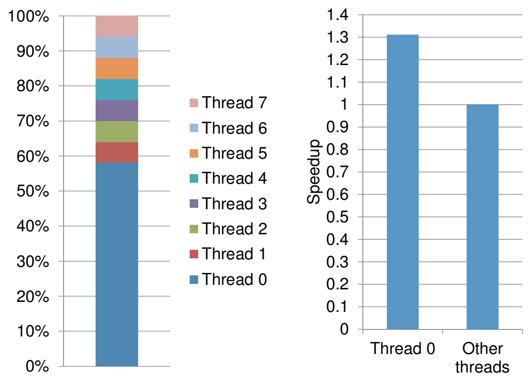


**Figure 3.1:** BFS's criticality stack and total program speedups from accelerating the identified critical and non-critical threads.

which is a tool for providing insights into an application's scaling behavior on multi-core hardware.

In the remainder of this chapter, we discuss those three new methods, explain how they are constructed, and show examples. Criticality stacks are discussed in Section 3.2, bottle graphs in Section 3.3, and speedup stacks in Section 3.4 and 3.5. Finally, we discuss related work in Section 3.6.

## 3.2 Criticality Stacks

Our first method for multi-threaded program analysis is the criticality stack. Criticality stacks show how critical each thread is to the performance of a multi-threaded application. For building criticality stacks we come up with a novel metric to measure thread criticality in parallel programs using synchronization behavior. This new criticality metric measures how much time a thread is performing useful work and how many threads are concurrently waiting. The metric gathers information for program execution intervals delineated by synchronization behavior. A thread has a larger criticality component when more threads wait concurrently on it, and thus it is more determinative of program running time.

Combining different threads' components into a criticality stack allows for easy comparison of parallel (im)balance. The criticality stack is a stacked bar graph that divides the program's total execution time

(100%) into each thread's criticality component. If all threads have approximately the same criticality, then no thread is critical, and no performance gain is to be expected by speeding up a single thread. If, however, certain threads have larger criticality than other threads, they reveal themselves as parallel bottlenecks.

We later validate criticality stacks by experimentally showing that speeding up the most critical thread (if one exists) results in significant performance speedups; accelerating identified non-critical threads on the other hand does not affect performance. Figure 3.1 illustrates this for the BFS benchmark: the criticality stack at the left shows that thread 0 is much more critical than all other threads. The graph at the right shows the program speedup when each of the threads is accelerated individually, running at twice the clock frequency. We present results for thread 0 and the maximum of all other threads (as they all result in no program speedup).

### 3.2.1 Constructing Criticality Stacks

A thread's criticality depends on both if it is doing useful work[1], and if other threads are waiting for it. We say a thread is critical if its progress at a certain point determines the progress of the whole program. One example is when all threads but one have reached a barrier. Because all other threads are waiting, the progress of the one thread that is still executing equals the progress of the whole program, and therefore this thread is critical.

In general, identifying the most critical thread in parallel programs is non-trivial. Figure 3.2 shows an example program with 4 threads that has both barrier (horizontal line across all threads) and critical section (darker vertical bar) synchronization. Thread 3 has the largest running time ($t_0 + t_1 + t_2 + t_3 + t_4 + t_5 + t_6$=17) and therefore performs most useful work; thread 0 on the other hand waits the longest for acquiring the critical section and keeps all other threads waiting at the barrier. It is not obvious which thread is most critical to overall performance.

To comprehensively compute thread criticality, we propose our criticality metric that takes into account both running time and number of waiting threads. Execution time is divided into a number of intervals.

[1]When a thread is spinning or busy waiting, we assume that it is not performing useful work. In the remainder of this work, we will denote a thread that is performing useful work as 'running', 'active', or 'executing', excluding spinning.

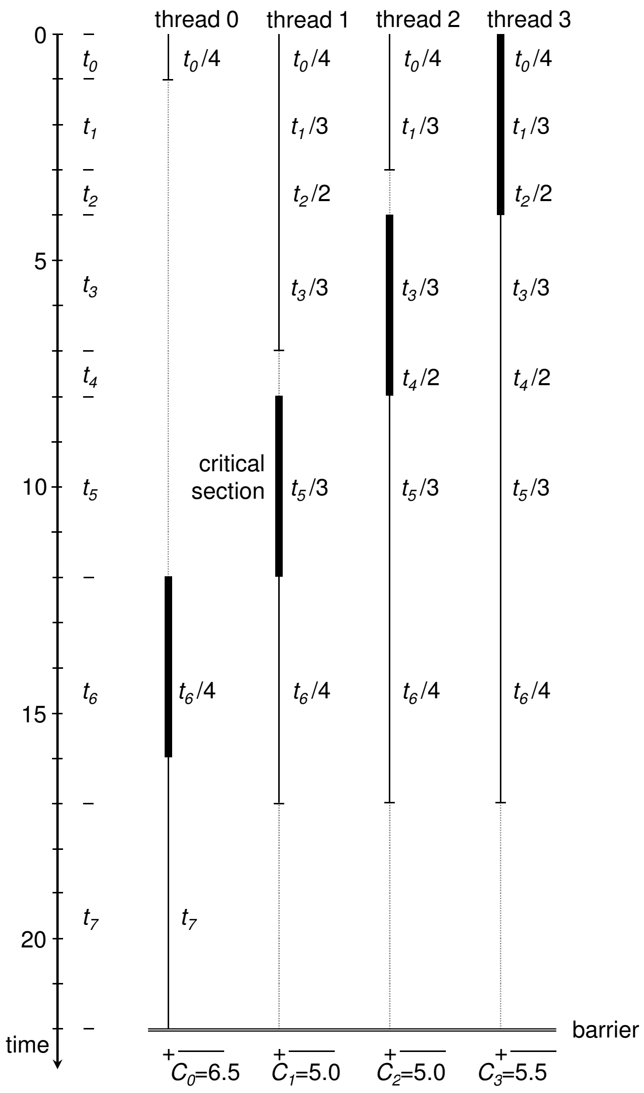


**Figure 3.2:** Criticality calculation example.

A new interval begins whenever any thread changes state, from active to inactive or vice versa, as a result of synchronization behavior. Each active thread's criticality number gets a portion of interval time $t$. In other words, time $t$ is divided by the number of threads doing useful work, and this is added to each thread's criticality sum (see Figure 3.2). This metric essentially weights time, adding more to active threads for which many threads wait, and less to active threads when no threads are waiting.

We formalize the criticality metric in the following way. Suppose that for a time interval $t$, $r$ out of $n$ threads are running. For the $r$ threads that are running we add $\frac{t}{r}$ to their respective criticality counter. For the other $n - r$ threads, we add nothing. In each interval, the set of running threads is fixed. Assume there are $N$ such intervals over the whole program (or a phase of the program), $t_i$ is the duration of interval $i$, $r_i$ is the number of running threads in that interval and $R_i$ is the set containing the thread IDs of the running threads (therefore $|R_i| = r_i$). Then the total criticality of thread $j$ equals

$$C_j = \sum_{i=0}^{N-1} \begin{cases} \frac{t_i}{r_i}, & \text{if } j \in R_i \\ 0, & \text{if } j \notin R_i \end{cases} \tag{3.1}$$

Figure 3.2 shows an example of how the criticality metric is calculated. Thread 0 has a total criticality of $t_0/4 + t_6/4 + t_7$=6.5. Threads 1, 2, and 3 all have lower criticality sums at 5, 5, and 5.5, respectively. Therefore, thread 0 is determined to be the most critical thread in this example. This might seem counter-intuitive because it has the smallest total running time ($t_0+t_6+t_7$=11) compared to all other threads (thread 1 = 16, thread 2 = 16, and thread 3 = 17). Accelerating thread 3 would reduce the execution time of the critical section, and as a result, threads 0, 1 and 2 would enter their critical sections sooner, however, thread 0 would still reach the barrier much later than the other threads, resulting in only a small speedup. Speeding up thread 0 on the other hand results in a much larger speedup, because it is guaranteed to reduce the barrier waiting time of all other threads, so thread 0 is indeed more critical as detected by the criticality metric. By taking into account the number of active threads, our metric clearly illustrates differences in criticality between threads.

An important characteristic of this metric is that the sum of all threads' criticalities equals the total execution time. Formally, if $T$ is the total execution time of the parallel program (or a phase), then

$$\sum_{j=0}^{n-1} C_j = T. \tag{3.2}$$

This is intuitive, as for every interval $r$ times $\frac{t_i}{r}$ is accounted, which gives a total of $t_i$ over all threads, and $\sum_{i=0}^{N-1} t_i = T$. This property allows us to divide each criticality sum by $T$ to obtain each thread's

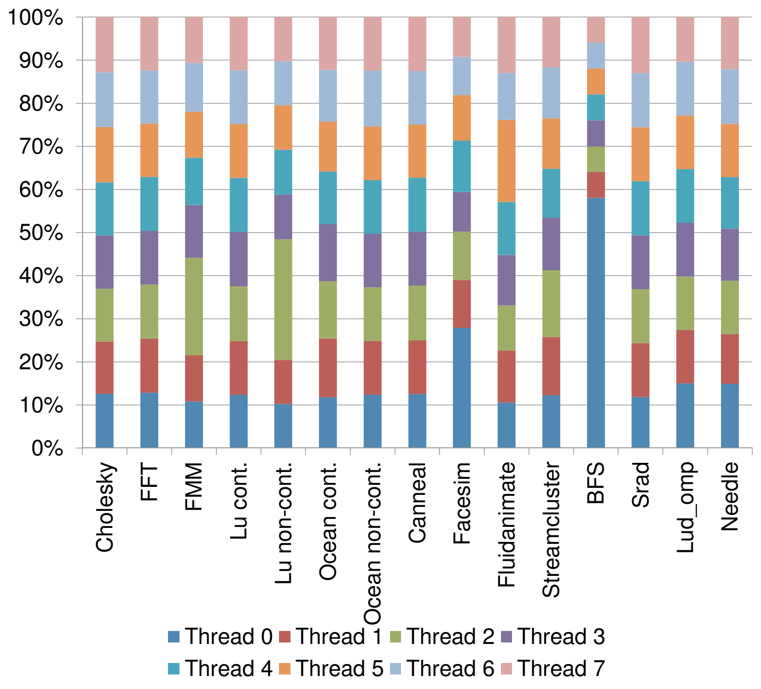


**Figure 3.3:** Criticality stacks for all benchmarks with 8 threads.

normalized criticality component. We represent these components in a stacked bar, yielding the criticality stack, which breaks up a program's total execution time into each thread's criticality percentage.

### 3.2.2 Example Criticality Stacks

Figure 3.3 shows the criticality stacks for a set of benchmarks when executed with 8 threads on an eight-core processor, with 100% of the execution time broken up into each thread's criticality percentage. (See Section 4.2 for an explanation about how we measure criticality stacks, and Section 4.3 for our experimental setup.) For some benchmarks, all criticality components are approximately equal-sized (Cholesky, FFT, Lu cont., Ocean cont., Ocean non-cont., Canneal, and Srad). There is no critical thread in these cases, meaning there is almost perfect parallel balance and thus speeding up any single thread will yield no performance gain. For the other benchmarks, one thread has a significantly larger fraction of criticality compared to the others, meaning that those benchmarks suffer from parallel imbalance: thread 2 for FMM, Lu non-cont., and Streamcluster; thread 0 for Facesim, BFS, Lud-omp, and Needle; and thread 5 for Fluidanimate. This is the most critical thread, and it is expected that speeding it up will result in a considerable performance gain, while speeding up other threads will have no significant performance impact. We will validate that this is indeed the case in Section 4.4.

## 3.3 Bottle Graphs

Bottle graphs extend upon criticality stacks by showing parallelism next to thread criticality, and its impact on performance. Just like criticality stacks, bottle graphs are stacked bar graphs. The height along the y-axis of a bottle graph is the total application execution time, see Figure 3.4 for an example. The stacked bar represents each thread as a box: the height is the thread's share of the total program execution time (and corresponds to thread criticality as described in the previous section); the width is the number of parallel threads that this thread runs concurrently with, including itself; and the box area is the thread's total running time. The center of the x-axis is zero, and thus parallelism is symmetric, reported on both the left and right sides of the zero-axis. We stack threads' boxes, sorting threads by their parallelism, with widest boxes (threads with higher parallelism) shown at the bottom and narrower boxes at the top of the total application bar graph — yielding a bottle-shaped graph, hence the name bottle graph.

Bottle graphs provide a new way to analyze multi-threaded application performance along the axes of execution time and parallelism. Bottle graphs visualize how scalable real applications are, and which threads have a larger total running time (total box area), which threads have limited parallelism (narrow boxes), and which threads contribute significantly to execution time (tall boxes). Threads that represent scalability bottlenecks show up as narrow and tall boxes around the 'neck' of the bottle graph. Bottle graphs thus quickly point software writers and optimizers to the threads with the greatest optimization potential.

The example bottle graph in Figure 3.4 represents a multi-threaded Java program, namely the DaCapo lusearch benchmark running with Jikes RVM on an 8-core Intel processor. This program takes 3.28 seconds to execute. There are 7 threads with visible bottle graph boxes, each having a different execution time share and different parallelism. The bottom four boxes represent application threads with a parallelism

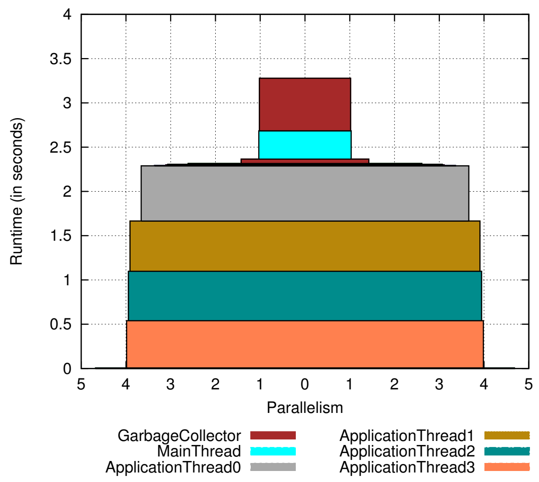


**Figure 3.4:** Example of a bottle graph: The lusearch DaCapo benchmark with 4 application threads, a main thread that performs initialization, and garbage collection threads running on Jikes JVM.

of approximately 4, and there is a main thread that performs benchmark initialization that has limited parallelism. There are two garbage collection (GC) threads, but the one with limited execution time share is a GC initialization thread, while the other GC thread that performs stop-the-world collection has a parallelism of only 1, because it runs alone.

Bottle graphs are an insightful way of visualizing multi-threaded program performance. Looking at the width of the boxes shows how well a program is parallelized. Threads that have low parallelism and have a large share in the total execution time appear as a large 'neck' on the bottle, which shows that they are a performance bottleneck. Bottle graphs thus naturally point to the most fruitful directions for effectively optimizing a multi-threaded program.

### 3.3.1 Constructing Bottle Graphs

For constructing bottle graphs we need to quantify the two dimensions of each thread's box in the graph, the height and the width, representing the thread's share in the total execution time and parallelism, respectively.

### Quantifying a Thread's Execution Time Share

Attributing shares of the total execution time to each of the threads of a multi-threaded program is not trivial. One cannot simply take the individual execution times of each of the threads, because their sum is larger than the program's execution time due to the fact that threads run in parallel. Individual execution times also do not account for variations in parallelism: threads that have low parallelism contribute more to the total execution time than threads with high parallelism. To account for this effect, we define the *share each thread has in the total execution time* (or *the height of the boxes*) as its individual execution time divided by the number of threads that are running concurrently (including itself). So, if $n$ threads run concurrently, they each get accounted one $n$th of their execution time. This is the same definition as the criticality metric for criticality stacks (see Section 3.2).

For completeness we repeat the definition of the criticality metric. Assume $t_i$ is the duration of interval $i$, $r_i$ is the number of running threads in that interval, and $R_i$ is the set containing the thread IDs of the running threads (therefore $|R_i| = r_i$). Then the total share (or criticality value) of thread $j$ equals

$$C_j = \sum_{\forall i: j \in R_i} \frac{t_i}{r_i}. \tag{3.3}$$

The execution time share of a thread is the height of its box in the bottle graph. Therefore, the sum of all box heights, or the height of the total graph, is the total program execution time. Unlike we did with criticality stacks, we do not normalize this number to 100% for bottle graphs.

### Quantifying a Thread's Parallelism

The other dimension of the graph – *the width of the boxes* – represents the *amount of parallelism of that thread*, or the number of threads that are co-running with that thread, including itself. A thread that runs alone has a parallelism of one, while threads that run concurrently with $n-1$ other threads have a parallelism of $n$. Due to the fact that the amount of parallelism changes over time, this number can also be a rational number.

The calculation of the execution time share already incorporates the amount of parallelism by dividing the execution time by the number of concurrent threads. We therefore define parallelism as the time a thread is active (its individual running time) divided by its execution time share. Formally, the parallelism of thread $j$ is calculated as

$$P_j = \frac{\sum_{\forall i:j \in R_i} t_i}{C_j} = \frac{\sum_{\forall i:j \in R_i} t_i}{\sum_{\forall i:j \in R_i} \frac{t_i}{r_i}}, \tag{3.4}$$

where $\sum_{\forall i:j \in R_i} t_i$ is the sum of all interval times where thread $j$ is active, which is its individual running time.

Equation 3.4 in fact calculates the weighted harmonic mean of the number of concurrent threads, i.e., the harmonic mean of $r_i$ weighted by the interval times $t_i$. It is therefore truly the average number of concurrent threads for that specific thread. We choose harmonic mean because metrics that are inversely proportional to time (e.g., IPC or parallelism) should be averaged using the harmonic mean while those proportional to time (e.g., CPI) should be averaged using the arithmetic mean [35].

Another interesting result from this definition of parallelism is that the execution time share multiplied by the parallelism – $C_j \times P_j$ – equals the individual running time of the thread, or in bottle graph terms: the height multiplied by the width, i.e., *the area of the box*, equals *the running time of a thread*. If we consider the running time of a thread as a measure of *the amount of work a thread performs*, then we can interpret the area of a box in the bottle graph as that thread's work. Due to parallelism (the width of the box), a thread's impact on execution time (the height of the box) is reduced.

This bottle graph design enhances their intuitiveness – a lot of information can be seen in one visualization – and quickly facilitates targeted optimization. Reducing the amount of work (area) of a thread that has a narrow box will result in a higher total program execution time (height) reduction compared to reducing the work for a wide box. The impact of a thread on program execution time can also be reduced by increasing its parallelism, which increases the width of the box and therefore decreases its height, if the area (amount of work) remains the same.

### 3.3.2 Example Bottle Graphs

In this work we evaluate the performance of Java applications with bottle graphs running on top of the Jikes RVM. The reason for choosing

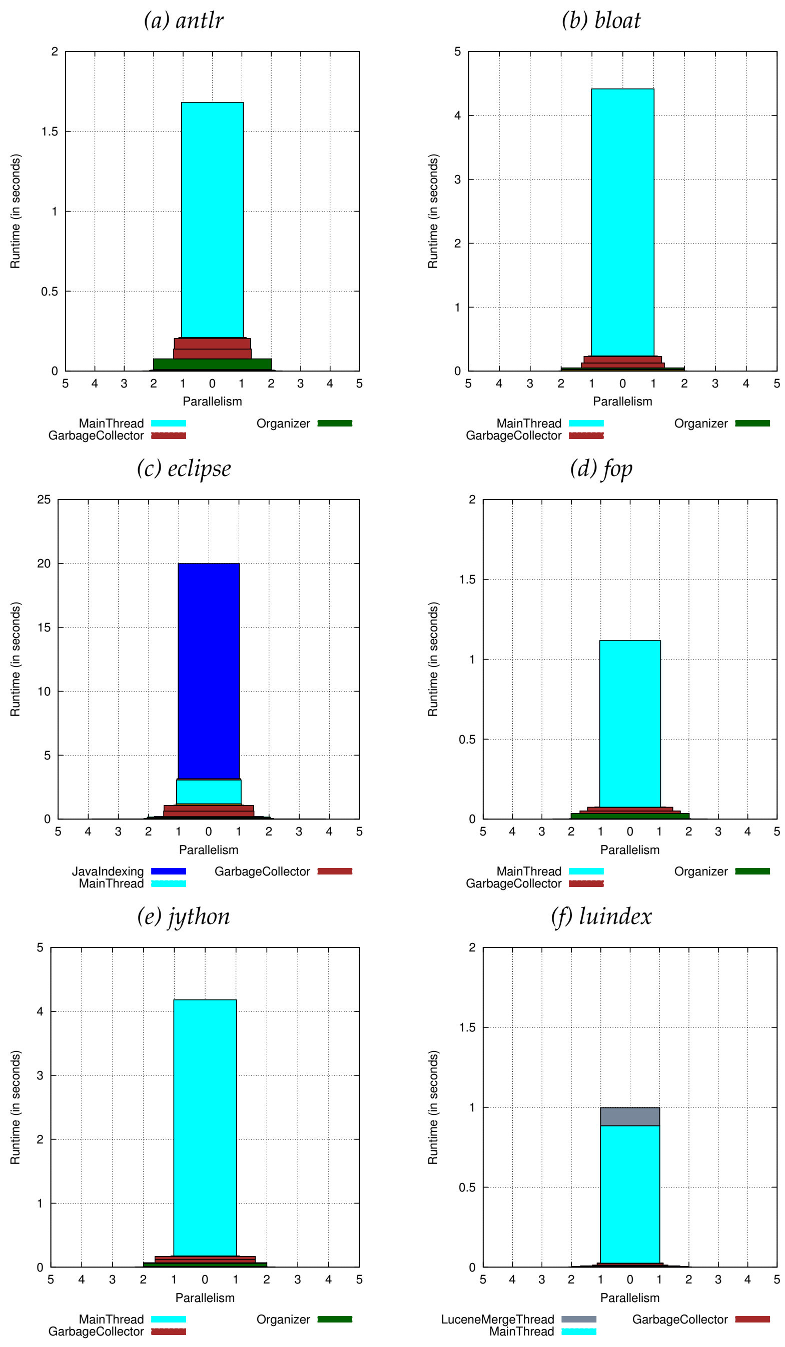


**Figure 3.5:** Bottle graphs for all single-threaded benchmarks with 2 GC threads.

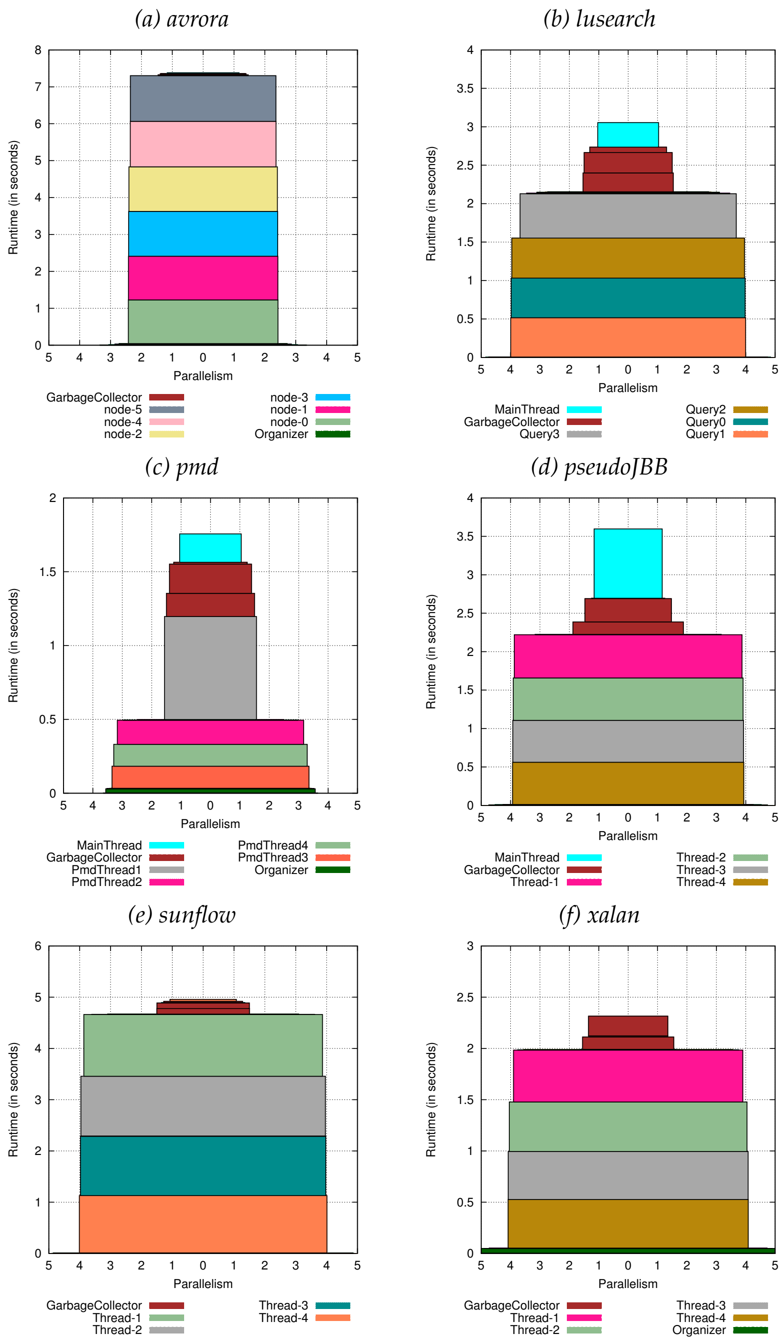


**Figure 3.6:** Bottle graphs for all multi-threaded benchmarks with 2 GC threads and 4 application threads.

this type of benchmarks is that Java applications are inherently parallel, because, apart from application threads, they also introduce service threads from the JVM. Those two types of threads behave differently and can interact with other, which makes this type of applications interesting for analysis.

Figures 3.5 and 3.6 show bottle graphs for single-threaded and multi-threaded Java benchmarks, respectively (only the threads that have a visible component in the bottle graph are shown). We discuss our profiling tool for generating bottle graphs later in Section 5.2 and the experimental setup in Section 5.3. All graphs have two garbage collector threads, and for the multi-threaded applications, we use four application threads. In general, turquoise boxes represent Jikes' MainThread which calls the application's main method. GC thread boxes are always presented in brown (including the GC controller thread, which explains the third GC box that appears in some graphs), while the dynamic compiler (called *Organizer*) is always presented in dark green. Application thread colors vary per graph. We found that all other JVM threads have negligible impact on execution time, and thus are not visible in the bottle graphs.

These graphs show the intuitiveness and insightfulness of bottle graphs. Single-threaded benchmarks can be easily identified as having a single large component with a parallelism of one, as can be seen from Figure 3.5. The graph of eclipse clearly shows that it behaves as a single-threaded application, as the JavaIndexing thread dominates performance, although it spawns multiple threads. Apart from the application and GC threads, antlr also has a visible Organizer thread, which is the only other JVM thread that was visible in all of our graphs. This thread has a parallelism of two, meaning that the JVM compiler always runs with one other thread, the MainThread in this case. Because it has a small running time, it does not have much impact on the parallelism of the main thread.

When we look at the multi-threaded benchmarks in Figure 3.6, we see that for lusearch, pseudoJBB, sunflow and xalan, the application threads have a parallelism of four, meaning that these benchmarks scale well to four threads. PseudoJBB has a rather large sequential component (in the MainThread), compared to the others. PseudoJBB does more initialization before it spawns the application threads, one per warehouse, to perform the work. Avrora is different in that it spawns six threads instead of four. This benchmark simulates a network of mi-

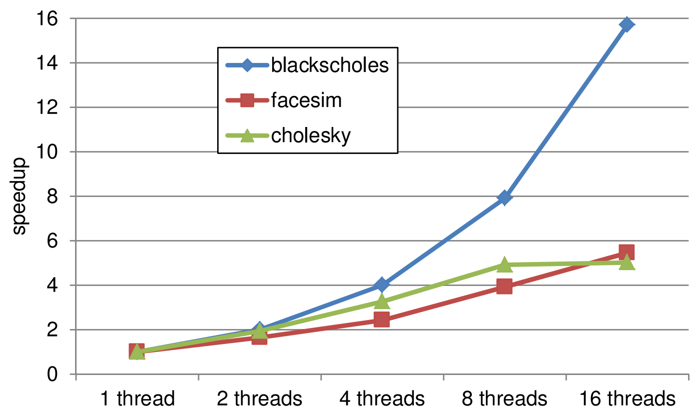


**Figure 3.7:** Speedup as a function of the number of cores for blackscholes, facesim (both PARSEC) and cholesky (SPLASH-2).

crocontrollers, and every microcontroller is simulated by one thread. Therefore, the number of threads spawned by avrora depends on the input, and the default input has six microcontrollers.

The bottle graph reveals that the parallelism of avrora's application threads is limited to 2.4, although there are six threads and 16 available hardware contexts. Avrora uses fine-grained synchronization to accurately model the communication between the microcontrollers, which reduces the exploited parallelism. This problem could potentially be solved by simulating close-by microcontrollers in one thread, instead of one thread per microcontroller, which will reduce the synchronization between the threads. Pmd is another interesting case: three of the four threads have a parallelism of more than three, but one thread has much lower parallelism and a much larger share of the execution time (PmdThread1). Pmd has an imbalance problem between its application threads.

## 3.4 Speedup Stacks

While criticality stacks and bottle graphs are great tools for analyzing the performance of individual threads of a multi-threaded program, both tools do not reveal how the application itself scales. A common way for understanding scaling behavior of an application is by looking at speedup curves, which report speedup as a function of the number of cores, as exemplified in Figure 3.7. Although a speedup curve gives a high-level view on application scaling behavior, it does not provide any insight with respect to why an application does or does not scale. There are many possible causes for poor scaling behavior, such as synchronization, as well as interference in both shared on-chip resources (e.g., last-level cache) and off-chip resources (e.g., main memory). Unfortunately, a speedup curve provides no clue whatsoever why an application exhibits poor scaling behavior.

Therefore, we propose our third tool, speedup stacks, which is a novel representation that provides insight into an application's scaling behavior on multi-core hardware. The height of the speedup stack is defined as $N$, with $N$ the number of cores or threads. The different components in a speedup stack define the actual speedup plus a number of performance delimiters: last-level cache (LLC) and memory interference components represent both positive and negative interference in the LLC and main memory; the spinning component denotes time spent spinning on lock and barrier variables; the yield component denotes performance deficiency due to yielding on barriers and highly contended lock variables; additional components are due to cache coherence, work imbalance and parallelization overhead. Figure 3.8 shows an example of a speedup stack. The intuition is that the scaling delimiters, such as negative LLC and memory interference, spinning and yielding, and their relative contributions, are immediately clear from the speedup stack. Optimizing the largest scaling delimiters is likely to yield the largest speedup, hence, a speedup stack is an intuitive and useful tool for both software and hardware optimizations.

### 3.4.1 Constructing Speedup Stacks

For explaining the key concept of a speedup stack, we refer to Figure 3.9. Because we want to compute a speedup stack from a single multi-threaded execution, we have to estimate many components. To simplify the discussion we focus on the parallelizable part of a program. Amdahl's law already explains the impact of the sequential part on parallel performance, hence, we do not consider it further in the remainder of this section. If of interest, including the sequential part in the speedup stack can easily be done (see Section 3.5.1).

We define $T_s$ as the execution time of (the parallelizable part of) a program under single-threaded execution. The execution time of the same program during multi-threaded execution will (most likely) be

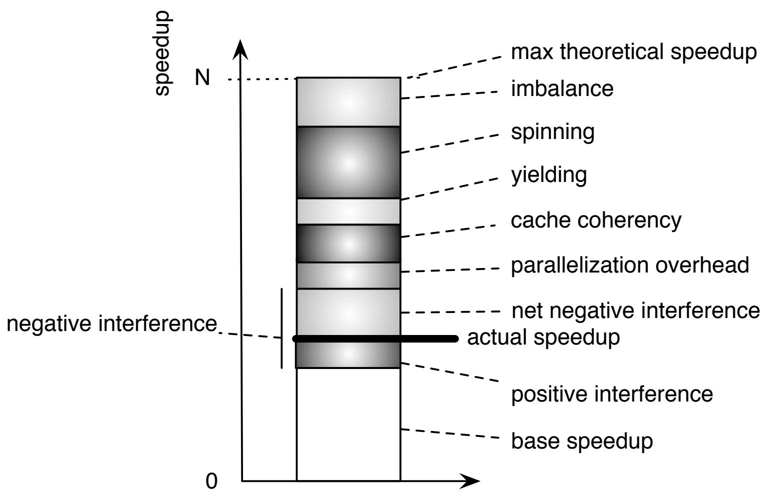


**Figure 3.8:** Illustrative speedup stack.

shorter, say $T_p$. We now break up the execution time of a thread during multi-threaded execution in various cycle components; note that the total execution time is identical for all threads under this break-up. The idealized multi-threaded execution time, assuming perfect parallelization, equals $T_s/N$ with $N$ the number of threads or cores. Note we use the terms 'thread' and 'core' interchangeably as we assume chip-multiprocessors, however, the concept of a speedup stack can also be applied to shared-memory multiprocessors (SMP) as well as simultaneous multi-threading (SMT) and other forms of multi-threading.

Obviously, the idealized multi-threaded execution time $T_s/N$ is not achieved in practice, hence multi-threaded execution time is typically longer, for a number of reasons. Parallelizing an application incurs overhead in the form of additional instructions being executed to communicate data between threads, recompute data, etc. This is referred to as parallelization overhead in the speedup stack. Other overhead factors include spinning, yielding and imbalance (threads waiting for other threads to finish their execution). Finally, there are interference effects in the memory hierarchy, both positive and negative in both the LLC and memory subsystem, as well as performance penalties due to cache coherence. Positive interference obviously offsets negative interference. In a rare case, positive interference could lead to superlinear speedups in case negative interference as well as the other overhead factors are small. Interference in a chip-multiprocessor is limited

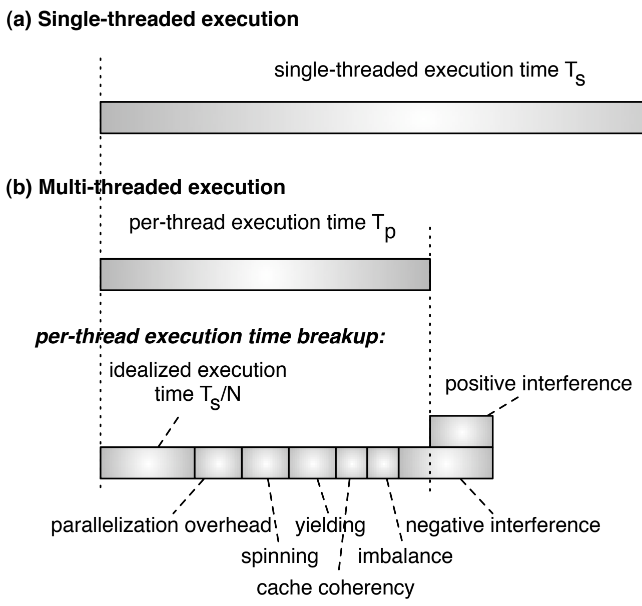

(a) Single-threaded execution
single-threaded execution time $T_S$
(b) Multi-threaded execution
per-thread execution time $T_p$
per-thread execution time breakup:
idealized execution time $T_S/N$
positive interference
parallelization overhead
spinning
yielding
cache coherency
imbalance
negative interference

**Figure 3.9:** Breaking up per-thread performance for computing speedup stacks.

to parts of the memory hierarchy; multi-threading architectures (e.g., SMT) also incur interference in the processor core, and hence an additional core interference component would need to be considered for these architectures.

Having broken up multi-threaded execution time into different cycle components for each thread, we revert to speedup, which is defined as single-threaded execution time divided by multi-threaded execution time:

$$S = \frac{T_s}{T_p}. \tag{3.5}$$

The way we build up a speedup stack is by profiling a multi-threaded execution, computing the aforementioned cycle components for each thread, and estimating single-threaded execution time from them. The way we estimate single-threaded execution time is essentially the reverse process of what we just explained considering Figure 3.9. We estimate the various cycle components during multi-threaded execution, and we subtract these cycle components from the measured execution time. This yields the fraction single-threaded execution time $\hat{T_i}$ for each thread. Summing these fractions $\hat{T_i}$ then provides an estimate for the total single-threaded execution time $\hat{T}_s$:

$$\hat{T}_s = \sum_i^N \hat{T}_i = \sum_i^N \left( T_p - \sum_j O_{i,j} + P_i \right), \tag{3.6}$$

with $O_{i,j}$ overhead component $j$ (negative interference, spinning, yielding, imbalance, cache coherence, parallelization overhead) for thread $i$, and $P_i$ positive interference for thread $i$.

Given the estimated single-threaded execution time, we can now estimate the achieved speedup $\hat{S}$:

$$\hat{S} = \frac{\hat{T}_s}{T_p} = \frac{\sum_i^N \hat{T}_i}{T_p} = \frac{\sum_i^N \left( T_p - \sum_j O_{i,j} + P_i \right)}{T_p}. \tag{3.7}$$

We now reformulate the above formula to:

$$\hat{S} = N - \sum_i^N \frac{\sum_j O_{i,j}}{T_p} + \frac{\sum_i^N P_i}{T_p}. \tag{3.8}$$

This formula immediately leads to the speedup stack by showing the different terms in the above formula in a stacked bar. The height of the bar equals the maximum achievable speedup, namely $N$. The various terms denote the aggregate overhead components across all threads, along with the aggregate positive interference component.

In summary, the speedup stack consists of the base speedup plus a number of components, see also Figure 3.8. *Base speedup* is defined as

$$\hat{S}_{base} = N - \sum_i^N \frac{\sum_j O_{i,j}}{T_p}, \tag{3.9}$$

and denotes the achieved speedup not taking into account positive interference. The *actual speedup* then is the base speedup plus the positive interference component. The other components highlight overhead components due to negative interference, cache coherence, parallelization overhead, spinning, yielding and imbalance. Note that the net negative interference is computed as the negative interference component minus the positive interference component.

An intuitive interpretation of a speedup stack is that it shows the reasons for sublinear scaling and hints towards the expected performance benefit from reducing a specific scaling bottleneck, i.e., the speedup gain if this component is reduced to zero. This can guide programmers and architects to tackle those effects that have the largest impact on multi-threaded multi-core performance. Also, because speedup stacks already incorporate positive interference, they can be easily extended to show superlinear scaling of an application.

### 3.4.2 Scaling Delimiters

In this section, we describe the five major speedup delimiters of multi-threaded workloads on multi-core hardware: resource sharing, synchronization, cache coherence, load imbalance and parallelization overhead. In case an application spawns more threads than there are hardware thread contexts, scheduling also has an effect on performance — this case is left out in this work though.

#### Resource Sharing

In Chapter 2 we already discussed the impact of resource sharing on the performance of threads running on a multi-core processor. In case of a multi-threaded application, resource sharing among the cores can either have a positive or a negative impact. A negative impact happens for example when sharing a cache among the cores, and threads evict data of other threads from the cache, which means that the latter will experience more misses than in isolated execution. On the other hand, positive impact happens when one thread brings data into the cache that is later used by the other threads.

#### Synchronization

Just like resource sharing in the hardware, we described in Chapter 2 that threads interact with each other due to synchronization between the threads. The most commonly used synchronization primitives are locks and barriers. While this synchronization is necessary for a correct execution, it also puts a limit on the achieved speedup of an application because it can cause threads to wait, thereby prolonging multi-threaded execution time.

### Cache Coherence

Cache coherence ensures that private caches (e.g., the L1 caches in a CMP) are consistent with respect to shared data. Cache coherence introduces extra traffic on the bus or interconnection network, and causes additional misses when local cache lines that are invalidated through upgrades by other cores, are re-referenced later. Unnecessary cache coherence traffic may result from false sharing.

### Load Imbalance

Load imbalance means that one or a few threads need (substantially) more time to execute than the other threads, which puts a limit on the achievable speedup, as the execution time of a multi-threaded application is determined by the slowest thread. Load imbalance can be caused by an uneven division of the work among the threads, but it can also be a result of the impact of resource sharing, cache coherence or synchronization. A thread that has an equal amount of work compared to the other threads, but is delayed more through resource sharing, cache coherence or synchronization can become the slowest thread, limiting an application's overall performance.

Synchronization effects through barriers could also lead to load imbalance: an application should be developed such that all threads reach the common barrier at the same time; if not, load imbalance may cause some threads to wait for other threads to reach the barrier. In this work, we classify work imbalance at barriers as a synchronization effect.

### Parallelization Overhead

Parallelizing a program typically incurs some overhead. Threads need to be spawned, locks need to be checked, acquired and released, local calculations possibly need to be done multiple times if it is too costly to communicate their results via shared variables, etc. Because these extra instructions incur computation time, they contribute to the sublinear speedup effect.

Although parallelization overhead is very hard to measure during multi-threaded execution (because it results into additional instructions being executed and can only be quantified with a second single-threaded execution of the application), it is most visible to the programmer. A software developer has a good understanding of the

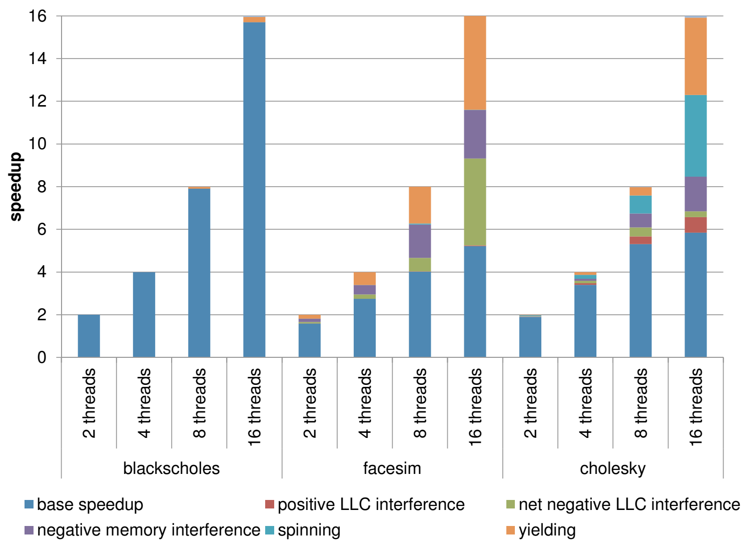


**Figure 3.10:** Speedup stacks as a function of the number of threads for blackscholes, facesim and cholesky.

amount of parallelization overhead. For speedup stacks, we do not measure the impact of parallelization overhead though (because we only do a multi-threaded execution and estimate single-threaded behavior from this). This implies that the estimated speedup is higher than the actual speedup in most cases.

### 3.4.3 Example Speedup Stacks

Figure 3.10 shows speedup stacks for the blackscholes, facesim and cholesky benchmarks for 2 up to 16 threads/cores (how we construct those stacks will be explained later in Section 6.2.1, and the experimental setup in Section 6.2.2); note that Figure 3.7 earlier in this chapter showed speedup curves for the same set of benchmarks. The blue bottom component represents the base speedup, i.e., the speedup without positive interference effects, or the number of threads minus all negative interference components. The red component on top of it (if present) represents the positive LLC interference component, hence the actual speedup is the sum of the blue and red components. The green component is the net negative interference LLC sharing component, i.e., the negative LLC sharing component minus the positive LLC sharing component. In other words, the negative LLC interference component equals the sum of the red and green components. If all negative cache sharing could be removed, then the speedup would increase with an amount proportional to the negative cache sharing component, which is the sum of the red and green components. The other components represent interference in the memory subsystem, spinning and yielding. The speedup stacks in Figure 3.10 do not show an imbalance component: as we measure the stacks over the entire parallel fraction of the program (i.e., between the divergence and convergence of the threads), the imbalance component is zero or nearly so, hence it is not visible.

As was apparent from the speedup curves in Figure 3.7, blackscholes shows almost perfect scaling; there are no significant scaling bottlenecks as clearly observed from the speedup stacks in Figure 3.10. Figure 3.7 also showed that speedup scales poorly with the number of threads and cores for facesim and cholesky. Although their speedup curves look similar and both benchmarks achieve comparable speedups, the speedup stacks shown in Figure 3.10 reveal that the reason for the limited speedups differs across the two benchmarks. For facesim, the main scaling delimiters are yielding, negative LLC interference and interference in the memory subsystem. In contrast, spinning is the major scaling bottleneck for cholesky, followed by yielding and memory interference.

These examples clearly show the value of speedup stacks, as they reveal the major scaling bottlenecks, which vary across programs, even if the actual speedups are comparable. Moreover, identifying these scaling bottlenecks without speedup stacks would be challenging. Guided by the speedup stacks, programmers can try to reduce synchronization overhead if spinning or yielding is large, for example by using finer grained locks and smaller critical sections. If negative interference in the LLC or main memory is a major component for several important applications according to the speedup stacks, processor designers can put more resources towards avoiding negative interference, for example through novel cache partitioning algorithms.

## 3.5 Speedup Stacks for Java

Speedup stacks, as we just proposed them, can be used for performance analysis of managed language applications like Java programs. However, managed runtime applications are different from native multi-threaded applications in the way that they exist of a combination of application and service threads that interact with each other during the execution of a program. This results in application threads that are waiting for various reasons. Figure 3.11 illustrates this: in the beginning the application threads wait because there is a sequential part in the application; during the execution the threads may need to wait due to synchronization; garbage collector may kick in; imbalance may occur among threads towards the end of the program execution. In our original speedup stacks, the waiting of threads (due to sequential code, synchronization, garbage collection, or imbalance) would be bumped together into a large yielding component.

However, since it is possible to know the reason why a thread is waiting, we can include this in our speedup stacks. This way the stacks reveal the interactions between service and application threads that are determinative to the execution time of a managed language program. It also leads to a new set of application scaling delimiters to include in the stack, namely garbage collection, sequential parts, thread imbalance, synchronization between threads, and hardware interference.

The way we build up these extended speedup stacks is similar to before. We start from Formula 3.8, but because we do not precisely quantify the impact of positive interference in these stacks, the formula becomes:

$$S = N - \sum_{i}^{N} \frac{\sum_j O_{i,j}}{T_p} \tag{3.10}$$

with $O_{i,j}$ as the scaling delimiter $j$ for thread $i$. To build a speedup stack, we first run the single-threaded version of the application to have a baseline. We then profile a multi-threaded execution of the application, computing the different overhead components (scaling delimiters) for each thread.

### 3.5.1 Scaling Delimiters for Java

In the next sections we discuss the different overhead components that $O_{i,j}$ is comprised of, and how they are integrated in Formula 3.10. We consider garbage collection, sequential parts of the application, synchronization, thread imbalance, and additional overheads as components in our speedup stacks.

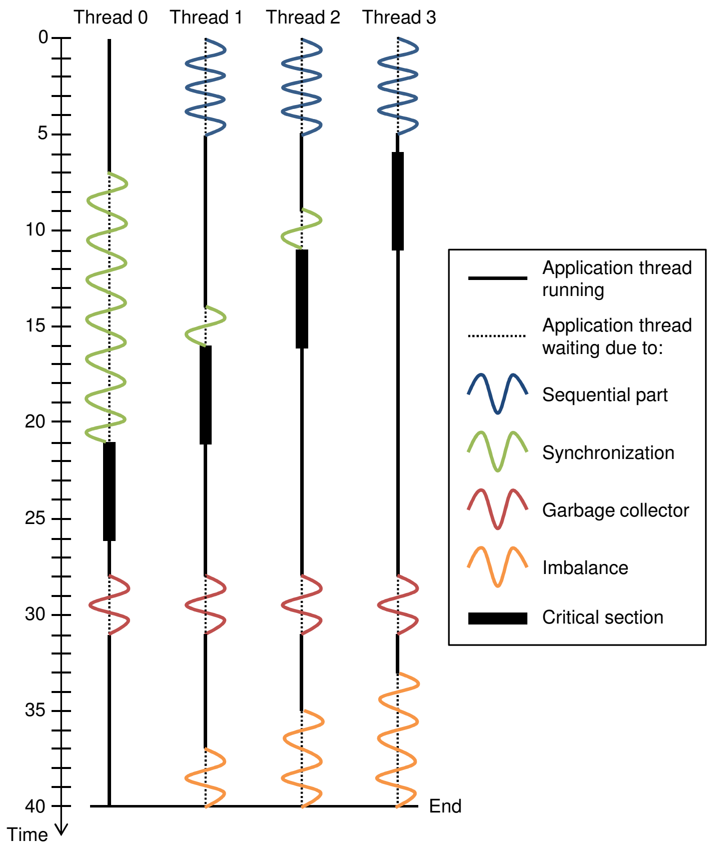


**Figure 3.11:** Example of applications threads in a managed language environment.

## Garbage Collection

Garbage collection (GC) is an integral component of many managed languages. Programmers benefit from the managed runtime environment automatically managing and collecting memory for them. However, this benefit comes at a cost; garbage collection does necessitate some space and time overhead. Previous work estimates that garbage collection takes on average 10% of an application's execution time [12]. This estimation assumes a high-performing stop-the-world generational garbage collector, meaning that the application is stopped while garbage collection traces the heap and reclaims memory. Concurrent collectors trace and reclaim memory concurrently with the application; however, they commonly require that the application stops in order for the GC to identify a consistent set of roots to trace from (including stack variables, statics and globals), and to finally reclaim memory back to the free list. Therefore, GC pauses the application, and thus affects its scalability and performance, and consequently should be included in a speedup stack.

Because the actual speedup of the multi-threaded application over the single-threaded version already takes garbage collection time into account, our overhead component need to consider only the scalability of garbage collection. We thus compare the amount of time spent on GC in the multi-threaded execution, multiplied by the number of threads, to the time for GC in the single-threaded execution. Integrating garbage collection into Formula 3.10 leads to:

$$S = N - \frac{N \times T_{GC,MT} - T_{GC,ST}}{T_p} - \sum_{i}^{N} \frac{\sum_j O^r_{i,j}}{T_p} \qquad (3.11)$$

where $T_{GC,ST}$ and $T_{GC,MT}$ is the time needed to do garbage collection under single-threaded and multi-threaded execution (which is the same for all threads), respectively, and $O^r_{i,j}$ are the remaining overhead components $j$ for thread $i$. We subtract $T_{GC,ST}$ from the garbage collection overhead, because the single-threaded execution also has a garbage collection component, and the speedup is measured over the whole program, including garbage collection.

It is clear to see that if the stop-the-world phase of garbage collection is perfectly scalable (i.e, $T_{GC,MT} = \frac{T_{GC,ST}}{N}$), this overhead component of speedup stacks would reduce to zero. Thus this performance delimiter suggests the effect of limited GC scalability on achieved program speedup.

**Sequential Parts of the Application**

The speedup of a program is limited by the amount of sequential execution in the application [2]. In a Java application, these sequential parts happen because of initialization of the JVM and data, performing initial compilation, spawning of application threads, etc. Integrating these sequential parts into the formula for speedup stacks is similar to how garbage collection is included, because during the time that garbage collection threads pause the application, the application itself can not make progress. Most of these sequential parts that happen during multi-threaded execution, also exist during single-threaded execution. Therefore we include these sequential parts in the following way into the formula:

$$S = N - \frac{N \times T_{GC,MT} - T_{GC,ST}}{T_p} - \frac{(N-1) \times T_{seq}}{T_p} - \sum_{i}^{N} \frac{\sum_j O^r_{i,j}}{T_p} \tag{3.12}$$

In this Formula $T_{seq}$ is the time needed for the sequential parts, we have the factor $N - 1$ because $T_{seq}$ is the same under single-threaded and multi-threaded execution (as opposed to garbage collection).

**Synchronization**

As we discussed in Chapter 2, threads of a multi-threaded application synchronize with each other in order to achieve correct execution. This synchronization leads to waiting time for threads because they want to acquire a lock held by another thread, wait until another thread reaches a certain point in the execution, etc. Integrating this waiting time for threads due to synchronization in Formula 3.12 leads to (similar to the yielding component in our original speedup stacks):

$$\begin{aligned} S = N - \frac{N \times T_{GC,MT} - T_{GC,ST}}{T_p} - \frac{(N-1) \times T_{seq}}{T_p} \\ - \frac{\sum_i^N Sync_i}{T_p} - \sum_{i}^{N} \frac{\sum_j O^r_{i,j}}{T_p}. \end{aligned} \tag{3.13}$$

where $Sync_i$ is the waiting time due to synchronization for application thread $i$. Thus, if all threads' waiting time due to synchronization with other threads would go to zero, this speedup stack component would disappear, thus resulting in a higher achieved speedup.

**Thread Imbalance**

Thread imbalance is the same as what we previously called load imbalance in our original speedup stacks. This imbalance happens when application threads do not finish their execution at the same point in time. Incorporating thread imbalance leads to the following formula:

$$\begin{aligned} S = N - \frac{N \times T_{GC,MT} - T_{GC,ST}}{T_p} - \frac{(N-1) \times T_{seq}}{T_p} \\ - \frac{\sum_i^N Sync_i}{T_p} - \frac{\sum_i^N Imbalance_i}{T_p} - \sum_i^N \frac{\sum_j O_{i,j}^r}{T_p}. \end{aligned} \tag{3.14}$$

In this formula $Imbalance_i$ is the waiting time for application thread $i$ due to imbalance.

**Remaining overhead components**

The speedup $S$ when using formula 3.14 has an additional component $O_{i,j}^r$. The remaining overhead is due to other factors that limit scalability and performance when moving from single to multi-threaded applications on modern hardware. For example parallelizing a program typically incurs overhead due to additional instructions being executed. Secondly, hardware resource sharing between threads translates into larger running time of threads. Therefore, we include a final component that we call *hardware interference* in our speedup stacks that estimates $O_{i,j}^r$, accounting for these extra overheads and showing their impact on speedup.

### 3.5.2 Example Speedup Stacks for Java

Figure 3.12(b) shows an example of a speedup stack for Java, together with the corresponding bottle graph for this application in Figure 3.12(a). In Section 6.3.1 we will discuss how we construct those speedup stacks and our experimental setup will be discussed in Section 6.3.2. In this example we show the lusearch DaCapo benchmark with four application threads running on top of Jikes RVM. We use an eight-core processor with hyper-threading enabled, meaning that parallelism and speedup is not limited by the number of available hardware threads. The bottle graph for this application reveals that the application threads scale very well, their parallelism is almost equal to four and their time share (height of the boxes) is very similar. This means there is a balanced execution of the application threads. However, if we look at the speedup stack, we see that the application achieves a limited speedup of two, while four is the ideal speedup in this case. This is counterintuitive because the bottle graph suggests

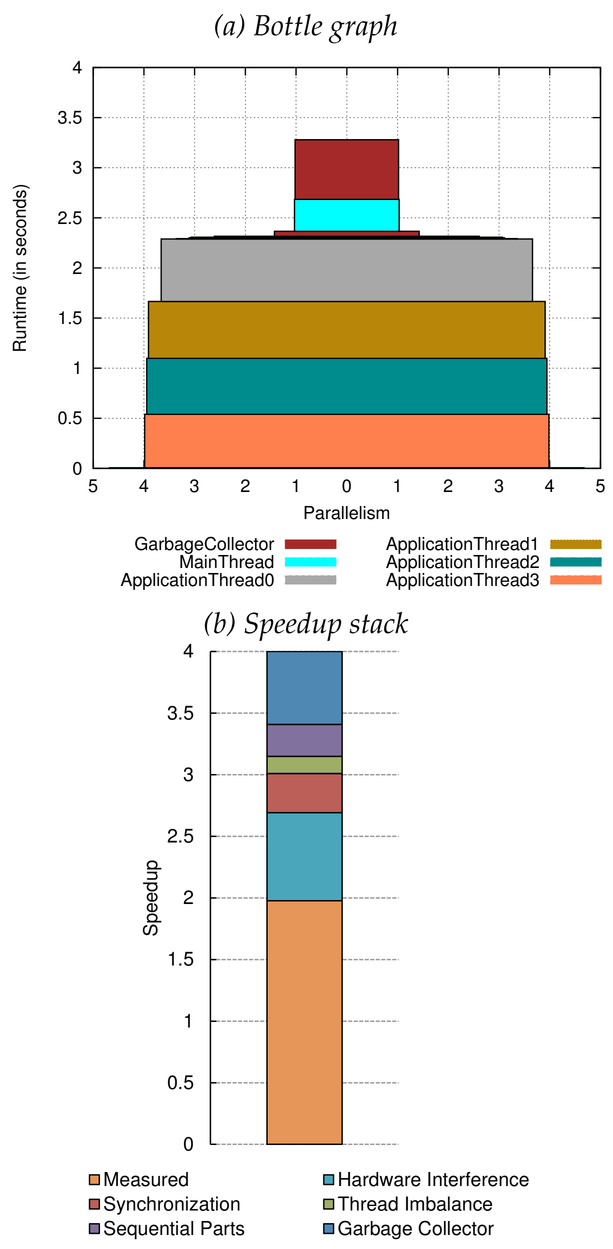


**Figure 3.12:** Example of a bottle graph and speedup stack: The lusearch DaCapo benchmark with 4 application threads, a main thread that performs initialization, and garbage collection threads running on Jikes JVM.

that the application scales well, apart from the garbage collector. This large box in the bottle graph for the garbage collector with limited parallelism, results in a significant part of the speedup that is lost for the application, as the speedup stack shows. Apart from the garbage collector, interference in the hardware also has a significant impact on speedup. This is not clear from the bottle graph, because this component translates into a longer execution time of threads. This example makes clear how speedup stacks are complementary to bottle graphs for understanding the performance of managed language applications.

## 3.6 Related Work

In this section, we describe related work in performance visualization and criticality analysis for multi-threaded applications.

### 3.6.1 Performance Visualization

Software developers heavily rely on tools for guiding where to optimize code. Commercial offerings, such as Intel VTune Amplifier XE [32], Sun Studio Performance Analyzer [33], and PGPROF from the Portland Group [58] use hardware performance counters and sampling to derive where time is spent, and point the software developer to places in the source code to focus optimization. Additional features offered include suggestions for potential tuning opportunities, collecting lock and wait behavior, visualization of running and waiting threads over time, etc. The key feature of these tools is that they provide fairly detailed analysis at fine granularity in small functions and individual lines of code. Recent work focused on minimizing the overhead even further, enabling the analysis of very small code regions, such as critical sections [15]. Other related work [37] proposes a simple and intuitive representation, called Parallel Block Vectors (PBV), which map serial and parallel phases to static code. Other research proposes the Kremlin tool, which analyzes sequential programs to recommend sections of code that would get the most speedup from parallelization [28]. All of these approaches strive at providing fine-grained performance insight. Unfortunately, none of these approaches provide a simple and intuitive visualization and understanding of gross performance scalability bottlenecks in multi-threaded applications, which is needed by software developers to guide optimization.

CPI stacks [21] are frequently used for identifying performance bottlenecks in single-threaded applications. Eyerman et al. [24] propose a cycle accounting architecture for constructing CPI stacks on out-of-order processors which is challenging to do given overlap effects among miss events and useful computation. CPI stacks are widely used for guiding software and hardware optimization for single-threaded applications but are less useful for multi-threaded applications because they do not incorporate synchronization between threads. One could argue that the speedup stack in the multi-threaded application domain is what the CPI stack is for single-threaded applications.

### 3.6.2 Criticality Analysis

Understanding program criticality is challenging because of various interaction and overlap effects across concurrent events, be it instructions or threads. Fields et al. [27] and Tune et al. [61] proposed offline techniques to analyze instruction criticality and slack based on data and resource dependencies in sequential programs. Li et al. [42] extended this offline approach to shared-memory programs. Hollingsworth [31] proposed an online mechanism to compute the critical path of a message-passing parallel program. Saidi et al. [55] use critical path analysis to detect bottlenecks in networking applications. Bhattacharjee and Martonosi [4] detect thread criticality in barrier-synchronized parallel programs by correlating criticality to cache misses. More recently, Cheng and Stenström [14] propose an offline analysis to detect critical sections on the critical path. None of this prior work addressed thread criticality in parallel, shared-memory programs with general synchronization primitives (including critical section, barrier and pipelined synchronization) as criticality stacks do.

## 3.7 Summary

In this chapter we introduced three new methods for analyzing performance and scalability of multi-threaded applications. Criticality stacks use a novel criticality metric and break down execution time into shares for each thread, facilitating detailed analysis of parallel imbalance. Bottle graphs represent each thread as a box, showing its execution time share (height) and parallelism (width), revealing exactly how scalable each thread is. Speedup stacks visualize achieved speedup of an application and the various scaling delimiters as a stacked bar. We presented two versions of speedup stacks, one that incorporates the impact of resource sharing on performance in a very detailed manner and a second version that is targeted towards managed runtime applications like Java programs. This second version provides less detailed information on the impact of resource sharing but provides new insights into how the interaction between application and virtual machine service threads affect program performance.

In the next three chapters we further discuss criticality stacks (Chapter 4), bottle graphs (Chapter 5) and speedup stacks (Chapter 6).

Chapter 4

# Criticality Stacks: Identifying Critical Threads

*In this chapter we analyze multi-threaded programs using criticality stacks, which is our performance analysis method for finding parallel imbalance between threads of a multi-threaded application.*

## 4.1 Introduction

While synchronization between threads is necessary, it also results in threads waiting for each other, as discussed in Chapter 2. This waiting of threads leads to an imbalance between threads, because not all threads make an equal progress. Therefore, some threads can be more critical to performance than others. For detecting this parallel imbalance and identifying the critical threads of an application, we use criticality stacks. Criticality stacks are build using our novel criticality metric that measures thread criticality in a parallel program using synchronization behavior.

In this chapter, we first present a hardware implementation for dynamically measuring thread criticality at a low overhead. For obtaining confidence in the accuracy of our criticality metric, we then validate the metric by experimentally speeding up threads. This experiment shows that accelerating threads that are identified as critical, results in a significant speedup for the application, while accelerating non-critical threads results in an unchanged performance of the application. We also compare our metric against closely related work that tries to address load imbalance caused by barriers [4]. We reimplement their technique that predicts thread criticality based on cache misses, and find that our criticality metric more accurately identifies the thread most critical to running time. Finally, we demonstrate how criticality stacks can help programmers for addressing performance problems.

## 4.2 Constructing Criticality Stacks

For constructing criticality stacks we need to dynamically measure thread criticality. This requires to determine at every moment in time how many threads are performing useful work. We now first detail how to identify which threads are active, which also delineates time intervals. We then describe our dedicated hardware implementation for calculating thread criticality in an efficient manner using very little extra energy and without interfering with the running program.

### 4.2.1 Identifying Running Threads

There are two main causes why a thread is not performing useful work: either it is scheduled out by the operating system, or it is spinning (using a waiting loop, constantly checking the synchronization variable). The operating system can easily communicate when it schedules threads in and out. Spinning is more difficult to detect, since the thread is executing instructions, albeit useless ones.

Either software or hardware can detect spinning. Software solutions involve adding extra instructions that denote spinning threads. These extra instructions are typically inserted in threading libraries (e.g., Pthreads) so that programmers do not have to explicitly add them. Hardware solutions use tables in the processor to keep track of backward branches [43], which possibly belong to a spinning loop, or repetitive loads [60], which are possibly loading a condition variable. Spinning is detected if specific conditions are met, i.e., no architectural state changes since the last branch or an update from another core to the repetitive load's address.

Both approaches have their advantages and disadvantages. A hardware solution can detect all types of spinning, including user-level spinning. On the other hand, a hardware solution detects spinning later (e.g., only after a certain threshold is reached), which can have an impact on the effectiveness of the technique that needs spinning information, and there is a chance to have false positives (e.g., a non-captured architectural state change) or false negatives (e.g., when the number of spinning iterations is under a certain threshold).

Software solutions on the other hand use semantic information from the program itself, and will only detect true spinning loops. Of course, user-level spinning that is not instrumented cannot be detected. However, if correctly instrumented, software accurately detects the start of the spinning, and can immediately indicate the end of the spinning.

For this study we use a software solution, since software detects spinning in a more timely manner and is easier to implement. The benchmarks we evaluate only use threading libraries to perform synchronization (Pthreads and OpenMP). We instrument all Pthread and OpenMP primitives that involve spinning (locks, barriers and condition variables). When the program enters and exits a spinning loop, we insert a call-down to notify hardware that the thread becomes inactive or active, respectively. The next section explains how hardware performs an online calculation of criticality based on these call-downs.

### 4.2.2 Calculating Criticality

To calculate the criticality metric as defined in Chapter 3, we need to know for each time interval which threads are performing useful work. To that end, we propose a small hardware component that keeps track of the running threads and the criticality of each thread. There is one criticality counter per thread (64 bit) and an 'active' bit that indicates whether the thread is running or not (see Figure 4.1). Each thread's criticality active bit is set or reset through the thread (de)activate calls that are sent from software. The cores or hardware contexts receive the calls from software, and send a signal to update the criticality state. These signals coming from the cores can either be transmitted over dedicated lines (a single line is sufficient for setting one bit), or through the existing interconnection network. In both cases, they do not incur much overhead, because the signal is only one bit and is sent relatively infrequently (we discuss frequency later in this section).

In addition to the per-thread counters and active bits, there is a counter that holds the number of active threads and a timer (see the bottom of Figure 4.1). The active thread counter is simply incremented when an activate call is received, and is decremented when a thread de-

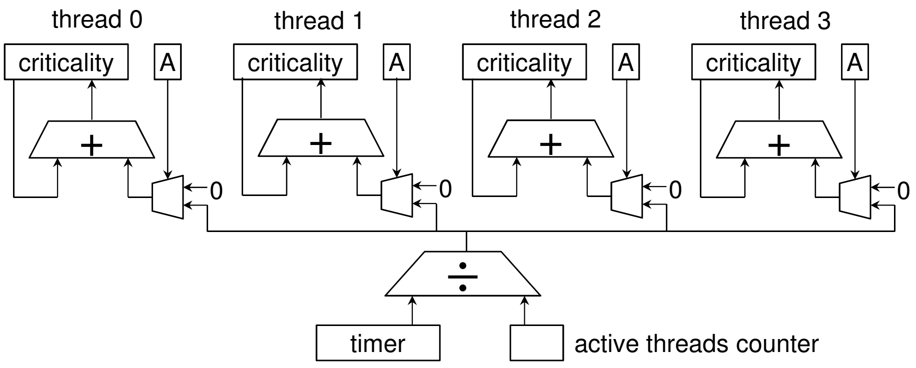


**Figure 4.1:** Hardware device for online criticality calculation ('A' is the active bit per thread).

activates. The timer keeps track of absolute time (hence it is independent of a core's frequency) since the previous synchronization event and is reset whenever an activate or deactivate call is received. Initially when a software call is received, the timer holds the duration of the past interval. Thus, before updating state, we add the result of the timer divided by the active thread counter to each thread's criticality counter for which the active bit is set. Then, the active bits and counter are updated and the timer is reset, indicating the start of a new time interval.

While conceptually we need a counter per thread, we can implement one counter per core or hardware context in reality (even when there are more threads than hardware contexts). Only while threads are running do their criticality counters need to be updated (inactive threads do not receive criticality anyway); thus, keeping one hardware counter plus active bit per core or hardware context allows running threads to update their criticality state. Upon a context switch, the operating system saves the criticality state for the thread being scheduled out, and initializes the core or context's criticality state to that of the thread becoming active. Thus, our implementation works with more threads than cores.

The advantage of using a dedicated hardware component is that it has negligible impact on the performance of a running application. The application just sends the (asynchronous) activate/deactivate calls and can continue its execution without waiting for an answer. In terms of hardware overhead, we need 65 bits per thread (a 64-bit timer plus the 'active' bit). For sixteen threads, this amounts to a total of 1,108 bits. Additionally, we need one integer divider (the interval duration is usually much larger than the number of threads, so the fraction after the decimal point can easily be ignored), and one 64-bit adder per thread. (Note the divider and adders can be low-performance, low-power units because they are off the processor's critical path.) In other words, the hardware overhead for computing criticality stacks is limited.

To calculate the power overhead, we recorded the number of updates per 10 ms time slice. For 16 threads, there are 1,920 updates per time slice on average, with a maximum of 31,776 updates. On every update, we need to perform an integer division and at most 16 additions (assuming 16 threads). According to Wattch [10], an integer division consumes approximately 0.65 nJ and an addition consumes 0.2 nJ in a 100 nm chip technology; energy consumption is likely to be (much) lower in more recent chip technologies, hence these estimates are conservative. This implies a maximum of 3.85 nJ per update, and by taking into account the number of updates per unit of time, this leads to an average 7.39 $\mu$W power consumption, and 0.12 mW at most, which is very small compared to the power consumed by modern-day high-end processors (around 100+ W).

## 4.3 Experimental Setup

We conduct full-system simulations using gem5 [6]. Table 4.1 shows the configurations of the simulated multi-core processors. We consider eight- and sixteen-core processors, running eight- and sixteen-threaded versions of the parallel benchmarks, respectively. Each core is a four-wide superscalar out-of-order core, with private L1 and L2 caches, and a last-level L3 cache that is shared among cores. The OS that we run is Linux version 2.6.27; a thread is pinned onto a core to improve data locality and reduce the impact of context switching.

We consider benchmarks from the SPLASH-2 [63], PARSEC [5] and Rodinia [13] benchmark suites, see Table 4.2. We evaluate those benchmarks from the suites that correctly execute on our simulator for both eight and sixteen threads, and for which thread-to-core pinning could be done reliably (i.e., there is a unique thread-to-core mapping). The benchmarks were compiled using gcc 4.3.2 and glibc 2.6.1. Our experimental results are gathered from the parallel part of the benchmarks. Profiling starts in the main thread just before threads are spawned and ends just after the threads join (however, there is the possibility of se-

| | |
|---|---|
| no. of cores | 8, 16 |
| core type | 4-wide out-of-order |
| base frequency | 2 GHz |
| L1 D-cache | 64 KB, private, 2 cycles |
| L1 I-cache | 64 KB, private, 2 cycles |
| L2 cache | 512 KB, private, 10 cycles |
| L3 cache | 8 MB, shared, 10 ns |
| memory bus | 32 GB/s |
| memory access | 100 ns |

**Table 4.1:** Simulated multi-core processor configurations for criticality stacks.

| *Suite* | *Benchmark* | *Input* |
|---|---|---|
| SPLASH-2 | Cholesky | tk29.O |
| | FFT | 4,194,304 points |
| | FMM | 32,768 particles |
| | Lu cont. | 1024×1024 matrix |
| | Lu non-cont. | 1024×1024 matrix |
| | Ocean cont. | 1026×1026 ocean |
| | Ocean non-cont. | 1026×1026 ocean |
| PARSEC | Canneal | Simmedium |
| | Facesim | Simmedium |
| | Fluidanimate | Simmedium |
| | Streamcluster | Simmedium |
| Rodinia | BFS | 1,000,000 nodes |
| | Srad | 2048×2048 matrix |
| | Lud_omp | 512×512 matrix |
| | Needle | 4096×4096 matrix |

**Table 4.2:** Considered benchmarks for criticality stacks.

quential parts of code within this region). This approach factors out the impact of the trivial case of speeding up the sequential initialization and postprocessing parts of the program, and allows us to use criticality information to analyze the challenging parallel part of the program.

While our evaluation is limited to these programs, which have both critical sections and barriers, criticality stacks could also be useful for analyzing heterogeneous applications. The criticality stack for pipelined parallel programs can reveal the thread or pipeline stage that most dominates running time. Similarly, our criticality metric could reveal imbalances in a task stealing context as well. In addition, the criticality metric can be calculated for setups with more threads than cores.

## 4.4 Validation and Analysis

We now present criticality stacks for our parallel applications. We computed criticality for each thread of our benchmarks with 8 and 16 thread configurations, and present stacks that summarize thread criticality. We validate our criticality metric in the next section using frequency scaling of individual threads. We then compare our speedups to those achieved by scaling a thread identified to be critical by previous work that is based on cache misses. Finally, we show the variance when scaling over a range of frequencies.

### 4.4.1 Validation of Criticality Stacks

Figure 4.2(a) shows the criticality stacks for the benchmarks when executed with 8 threads on 8 cores.We already discussed these criticality stacks in the previous chapter in Section 3.2.2. We now evaluate the validity of criticality stacks by checking that accelerating the most critical thread (if one exists) results in program speedups. Each simulation speeds up one thread by raising the core's frequency from 2 GHz to 4 GHz[1], and we present speedup results versus a baseline of all threads at 2 GHz in Figure 3.3(b) for 8 threads. For each benchmark we present the speedup obtained by accelerating each of the three threads that have the largest components in the criticality stack. For the other threads, the speedup was equal to or lower than the speedup of the third largest component.

Figure 4.2(b) shows that for the benchmarks that have equal-sized components in the criticality stack (see Figure 4.2(a)), there is no single thread that when accelerated results in a significant program speedup, which is in line with expectations. For the other benchmarks, speeding up the thread that has a significantly larger criticality than the other threads results in a considerable speedup for the whole program (e.g., Lu non-cont. and BFS have speedups over 20% and over 30%, respectively). Moreover, speeding up the thread with the largest component results in the largest speedup, while speeding up threads with smaller, roughly-equal components yields little or no speedup. One interesting phenomenon is Streamcluster, which has a few other threads besides thread 2 that have slightly larger criticality percentages, and thus each

[1]This frequency raise is not intended to resemble a practical situation, it serves only as a way to validate the criticality stacks.

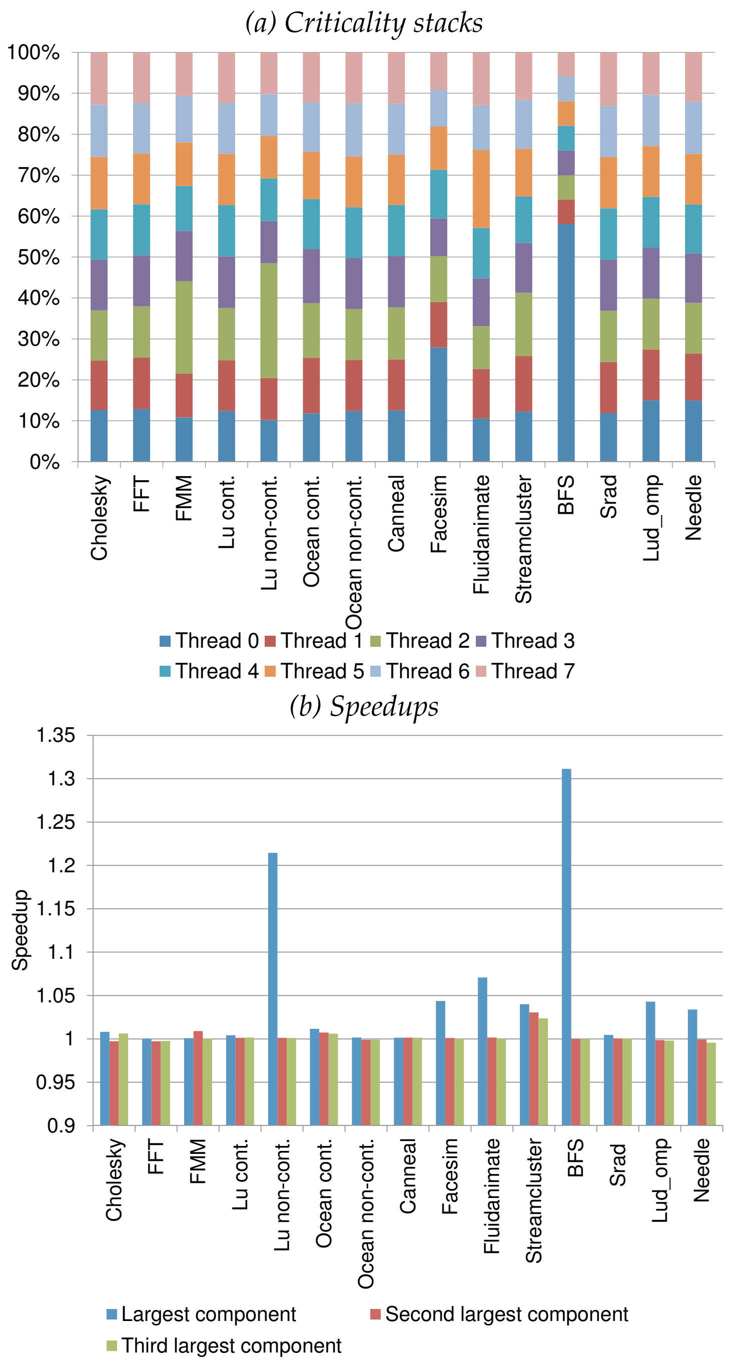


**Figure 4.2:** Criticality stacks for all benchmarks for 8 threads and corresponding speedups by accelerating one thread.

of the three threads show some speedup after being scaled up. This validates that criticality stacks provide useful and accurate information that can be used to guide optimizations.

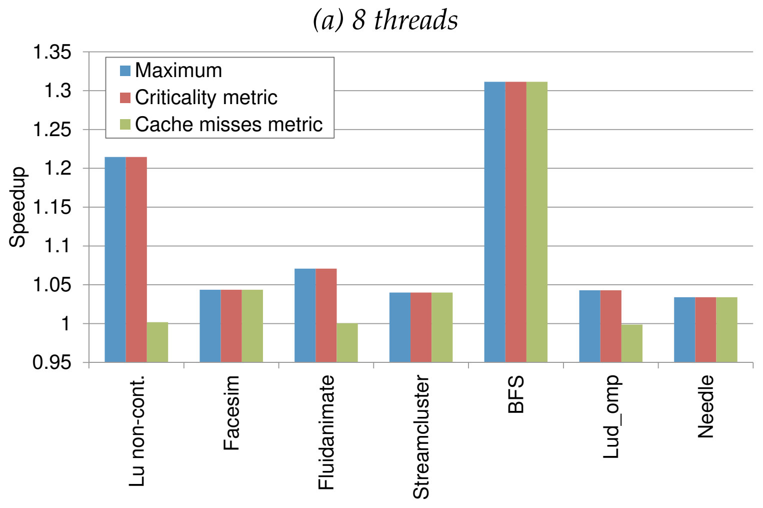
FMM is an exception to the rule because the criticality stack reveals that thread 2 is more critical than the others, but there is no speedup when this thread is accelerated. In fact, speeding up any single thread for this program never yields a significant speedup. Looking at the criticality stack after speeding up thread 2 revealed that that thread's component was reduced, but thread 7's component had grown significantly. FMM is an anomaly; for other benchmarks, speeding up the most critical thread resulted in a criticality stack with more equal-sized components. The criticality of the second thread for FMM is hidden, or overlapped, by the criticality of the first thread. Accelerating one thread just makes the other become more critical.

### 4.4.2 Comparison to Prior Criticality Metric

We compare the performance improvement of speeding up one thread that is identified as most critical for various ways of identifying the critical thread in Figure 4.3. We limit ourselves to accelerating one thread here because most of our benchmarks have only one most critical thread, but if more were detected, more threads could be accelerated. We present speedup results for the benchmarks that have a critical thread, i.e., speeding up a single thread results in a speedup of at least 3%. We present the results using a theoretical technique that takes the maximum speedup gained when accelerating each thread individually. We compare this with our criticality metric and with previous work that uses cache misses to define criticality [4]. The cache miss metric takes a weighted average of the number of L1, L2 and L3 cache misses[2], with the relative latency as a weighting factor.

Our newly proposed criticality metric achieves the same speedup as the maximum achievable speedup in all cases but one. For the 16-threaded version of Lu non-cont., there are two criticality stack components that are significantly larger than the others (thread 0 and thread 2). The maximum speedup is achieved by accelerating the second largest component (thread 2). A detailed analysis reveals that in the beginning of the program, thread 0 is executing alone for a while, spawning threads and distributing data, resulting in a large critical-

[2]We adapted the original formula in [4] to three levels of cache for our configuration.



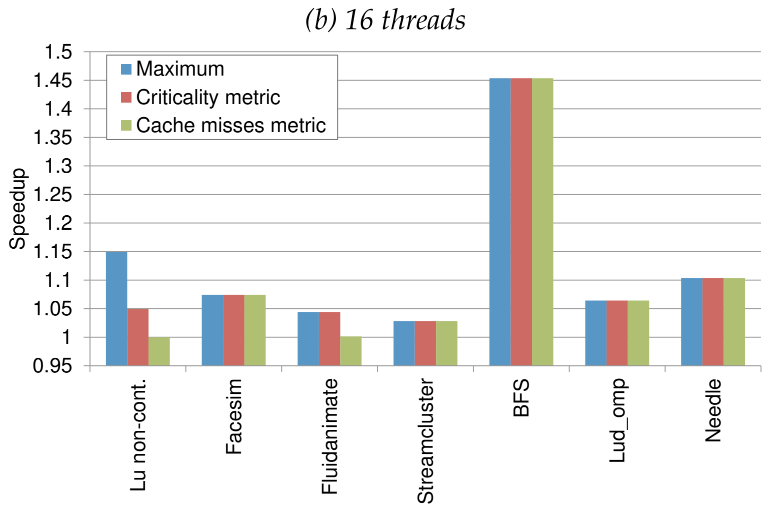


**Figure 4.3:** Comparison between our and a prior metric, and the maximum achievable speedup by accelerating one thread.

ity component. However, this process is very memory-intensive and results in many cache misses. Since the access time to memory is constant, raising the frequency of that core does not yield a significant speedup. After initialization, thread 2 becomes more critical, but its criticality does not exceed the accumulated criticality of thread 0. Although it is not the largest component, accelerating thread 2 yields the largest overall speedup.

Figure 4.3 reveals that using cache misses to identify critical threads is less accurate at identifying critical threads and does not lead to any performance gains for three benchmarks in the 8-thread configuration, and for two benchmarks with 16 threads, while our criticality metric always improves performance. The cache miss metric has been proven effective in barrier-synchronized parallel programs, while our new metric covers all types of synchronization. We conclude that our newly proposed metric is most effective at finding the thread most critical to performance.

### 4.4.3 Varying the Amount of Frequency Scaling

In the previous experiments, we raised the frequency of one thread from 2 GHz to 4 GHz. Now we explore more realistic frequencies between 2 GHz and 4 GHz, at increments of 0.25 GHz. We use our metric to find the most critical thread to speed up, and evaluate the impact of frequency scaling on the total program speedup, which reveals interesting insights about the applications. Figure 4.4 shows the resulting speedups for three representative benchmarks, and Figure 4.5 shows the criticality stacks for a subset of these frequencies.

These three benchmarks show different behavior as we scale frequency up. For Fluidanimate, Figure 4.4(a) shows that program speedup increases from 2 to 2.25 GHz, but remains constant when the frequency is raised further. This is a typical case of inter-thread synchronization criticality. Once the thread that other threads are waiting for is sped up enough such that the other threads do not have to wait anymore, no further speedup can be attained despite a faster core. This is also reflected in the change between the two criticality stacks on the left of Figure 4.5: after speeding up the most critical thread (thread 5), its criticality component shrinks, making the thread non-critical, and thus no further speedup can be obtained.

For BFS in Figure 4.4(b), the performance continues to improve as the frequency increases. BFS includes an inherently sequential part where only one thread is running, which continues to see performance improvements when sped up to higher and higher frequencies. When looking at the three criticality stacks for BFS on the right side of Figure 4.5, we see that after accelerating the most critical thread, this thread's component decreases, but remains the largest component.

In Figure 4.4(c), Lud_omp displays a mix of the behavior of the two previous cases: in the beginning the speedup raises considerably, while

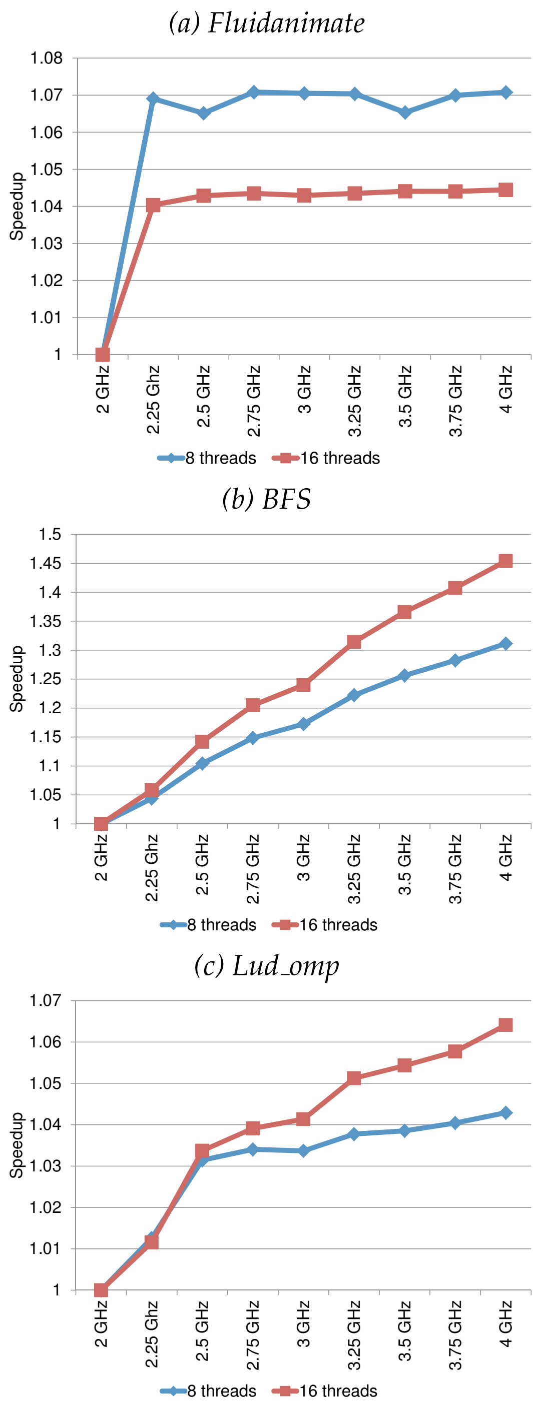


**Figure 4.4:** Impact of frequency scaling on achieved speedup.

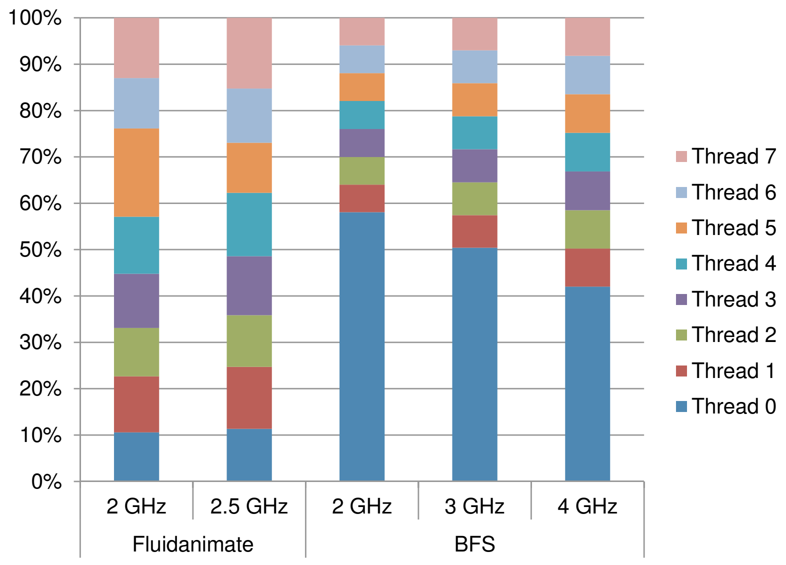


**Figure 4.5:** Impact of frequency scaling on criticality stacks.

after a certain frequency (2.5 GHz), speedup goes up at a slower pace. This benchmark's critical thread shows both inter-thread synchronization criticality and sequential criticality. For applications such as this, setting the frequency of the critical thread to the place where speedup slows, yields the best performance and energy consumption balance.

### 4.4.4 Steering Software Optimization

Having validated criticality stacks, we now consider a use case where we use criticality stacks to steer software optimization. When looking at the right side of Figure 4.5 we see that BFS suffers from excessive critical imbalance, even when the most critical thread is sped up to a high frequency. We investigated this benchmark further to determine whether, as predicted, there is some sequential part of the program that slows down progress. The main work of BFS, which does breadth-first search of a tree data structure, is performed in a `do-while` loop. Inside the loop are two `for` loops that loop over all of the nodes in the tree. Only the first is parallelized. The first loop visits the edges of each node, potentially updating data.

The second, unparallelized loop goes over each node of the tree, checking if it was updated. If there were updates, it sets the `do-while`

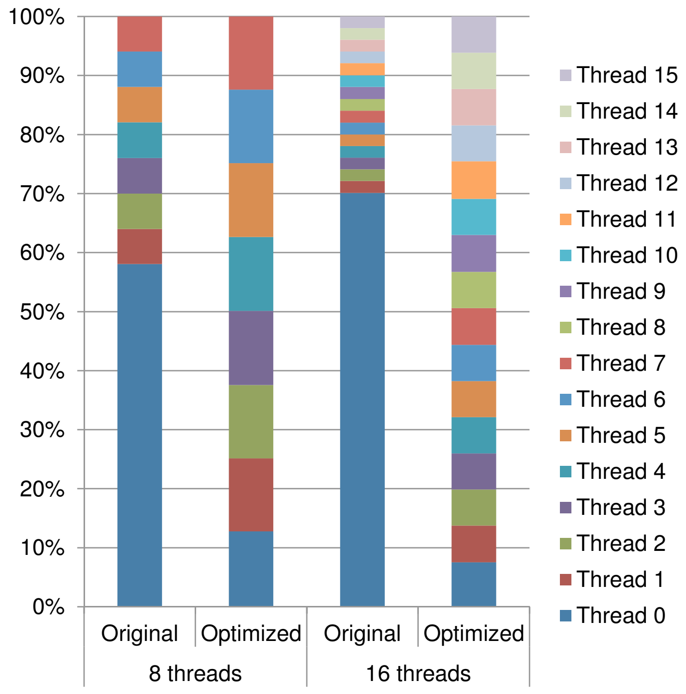


**Figure 4.6:** Example of using criticality stacks as a guide for software optimization (BFS benchmark).

flag to loop again, otherwise the `do-while` loop can terminate. We surmise that the most critical thread identified with our stacks, thread 0, is responsible for performing the second `for`-loop, which runs sequentially.

We analyzed the second loop, determined it has no dependencies between iterations, and optimized it by parallelizing the loop. After this small program change, Figure 4.6 presents the comparison between the unoptimized and optimized BFS criticality stacks, for both 8 and 16 threads. While Figure 4.5 shows that scaling to even large frequencies did not remove the criticality bottleneck, with software analysis and editing, we achieve balanced criticality stacks, as seen on the right in Figure 4.6. After this code change, BFS achieves a 1.67× and 2.16× speedup for 8 and 16 cores, respectively. These improvements are significantly better than the 31% and 45% speedups that are achieved through frequency scaling alone to 4 GHz (in Figure 4.4(b)). This use case illustrates that criticality stacks are a useful tool to assist software programmers in analyzing and fixing parallel imbalance.

## 4.5 Summary

After having introduced criticality stacks in the previous chapter, we validated them and used them to analyze parallel performance in this chapter. We describe a simple hardware design that takes a very small amount of power, while being off the processor's critical path, to compute criticality stacks during the execution of an application. We validate the accuracy and utility of criticality stacks by demonstrating that our low-overhead online calculation approach indeed finds the thread most critical to performance, improving over a previously proposed metric based on cache misses. We also showed how criticality stacks can be used to optimize software code. After optimizing the code of one benchmark based on criticality imbalance, we achieve an average speedup of $1.9\times$. In Chapter 7, we show how criticality stacks can be used to dynamically optimize performance of multi-threaded applications.

# Chapter 5

# Bottle Graphs: Visualizing Per-Thread Performance

*In this chapter we discuss our second performance analysis method, bottle graphs. Bottle graphs visually show per-thread scaling behavior of a parallel program.*

## 5.1 Introduction

In the previous chapter we discussed criticality stacks, which display per-thread contributions to total running time. Criticality stacks mainly focus on synchronization, and do not incorporate a notion of thread parallelism. Moreover, constructing criticality stacks requires hardware modifications and, while it points out thread imbalances, it does not suggest how much gain could be achieved by making a particular thread better able to co-execute with other threads. Therefore, we developed a second performance analysis method, bottle graphs.

Having introduced bottle graphs in Chapter 3, we now further discuss them in this chapter. We show how bottle graphs of real applications running on real hardware can be constructed through operating system support, by using light-weight Linux kernel modules. Using kernel modules, our bottle measurements incur very little overhead (0.68% on average), require no recompilation of the kernel, and require no modifications to applications or hardware. To demonstrate the power of bottle graphs as an analysis and optimization tool, we perform an experimental study of 12 single- and multi-threaded benchmarks written in Java, both from the DaCapo suite and pseudoJBB from SPEC, and running on top of Jikes RVM. We vary the number of application threads, number of garbage collection threads, analyze performance differences between benchmark iterations, and study the scalability of JVM service threads in conjunction with application threads.

## 5.2 Constructing Bottle Graphs

Just like criticality stacks, bottle graphs need to calculate a criticality value for each thread. Although we can construct bottle graphs with the hardware component that we designed for generating criticality stacks (see Section 4.2.2), we now propose a new method that is implemented in software, and measures the values in order to construct a bottle graph of an application running on an actual processor. The tool needs to detect:

1. The number of active threads, to calculate the $r_i$ values.
2. The IDs of the active threads, to know which threads should be accounted time shares.
3. Events that cause a thread to activate and deactivate, to delimit intervals.
4. A timer that can measure the duration of intervals, to get the $t_i$ numbers.

The operating system (OS) is a natural place to construct our tool, as it already keeps track of thread creation, destruction, and scheduling, and has fine-grained timing capabilities. We build a tool to gather the necessary information to construct bottle graphs using kernel modules with Linux versions 2.6 and 3.0. The kernel modules are loaded using a script that requires root privileges. Communication with the modules (e.g., for communicating the name of the process that should be monitored) is done using writes and reads in the /proc directory. We use kernel modules to intercept system calls that perform thread creation and destruction, that schedule threads in and out, and that do synchronization with futex (a Linux system call which implements thread yielding). Our tool keeps track of the IDs of active threads and the timestamp of interval boundaries. Our modules also keep track of two counters per thread: one to accumulate the running time of the thread (i.e., the total time it is active) and the other to accumulate the execution time share (i.e., the running time divided by the number of concurrent threads).

A kernel module is triggered upon a (de)activation call, and updates state and thread counters in the following way. The module obtains the current timestamp, and by subtracting the previous timestamp from it, determines the execution time of the interval that just ended. It adds that time to the running time counter of all threads that were running in the past interval. It also divides that interval's time by the number of active threads, and adds the result to the time share counters of the active threads. Subsequently, the module changes the set of running threads according to the information attached to the call (thread activation or deactivation, and thread ID), and adapts the number of active threads. It also records the current timestamp as the beginning of the next interval. When the OS receives a signal from software, the two counters for each thread are written out, and this information is read by a script that generates the bottle graphs.

There are several advantages of using kernel modules to measure bottle graph components:

1. The program under study does not need to be changed; the tool works with unmodified program binaries.

2. The modules can be loaded dynamically; there is no need to recompile the kernel.

3. We can make use of a nanosecond resolution timer, which enables capturing very short active or inactive periods. We used ktime_get and verified in /proc/timer_list that it has nanoscale resolution.

4. In contrast to sampling, our kernel modules continuously monitor all threads' states accurately, and aggregate metric calculations online without loss of information.

5. The extra overhead is limited, because the calculations are simple and need to be done only on thread creation and scheduling operations. On average, we measure an average 0.68% increase in program execution time compared to disabling the kernel modules, with a maximum of 1.11%, see Table 5.1 for per-benchmark overhead numbers.

**Discussion of design decisions.** In the design of our tool and experiments, we have made some methodological decisions, which do not limit the expressiveness of bottle graphs. We keep track of only synchronization events caused by futex, and thus our tool does not take into account busy waiting in spin loops, or interference between threads in shared hardware resources (e.g., in the shared cache and in the memory bus and banks). Threading libraries are designed to avoid long active spinning loops and yield threads if the expected waiting time is more than a few cycles, so we expect this to have no visible impact on the bottle graphs. Interference in hardware resources is a low-level effect that is less related to the characterization of the amount of parallelism in a multi-threaded program. When a thread performs I/O behavior, the OS schedules out that thread. Thus, we choose not to track I/O system calls in our kernel modules because most I/O behavior is already accounted for as inactive. Furthermore, we provide sufficient hardware contexts in our hardware setup, i.e., at least as many as the maximum number of runnable threads. We thus ensure that threads are only scheduled in and out due to synchronization events, and factor out the impact of scheduling due to time sharing a hardware context. If the number of hardware contexts were less than the number of runnable threads, the bottle graph's measured parallelism would be determined more by the machine than by the application characteristics.

In the majority of our results, we read out our cumulative thread statistics, including running time and execution time share, at the end of the program run. We then construct bottle graphs that summarize the behavior of the entire application execution on a per-thread basis. However, our tool can be given a signal at any time to output, and optionally reset, thread counters, so that bottle graphs can be constructed throughout the program run. Thus, bottle graphs can be used to explore phase behavior of threads within a program run (as we do in Section 5.4.4), or to analyze particular sections of code for scalability bottlenecks. Furthermore, while our OS modules keep track of counters per-thread, in the case of thread pools, the bottle graph scripts could be modified to group separate threads into one component, if desired. Both execution time share and parallelism are defined such that it is mathematically sound to aggregate thread components.

## 5.3 Experimental Setup

We perform experiments on unmodified applications running on real hardware to demonstrate the usefulness of bottle graphs. While bottle graphs can be used to analyze any multi-threaded program, in this work we chose to analyze Java applications. Not only are managed languages like Java widely used in the community, but they also provide the added complexity of runtime service threads that manage memory and do dynamic compilation alongside the application. Thus, we can analyze both application and service thread performance and scalability using our visualization tool.

We evaluate Java benchmarks, see Table 5.1, from the DaCapo benchmark suite [9] and one from the SPEC 2005 benchmark suite, pseudoJBB, which is modified from its original version to do a fixed amount of work instead of run for a fixed amount of time [57]. There are 6 single-threaded (ST) and 6 multi-threaded (MT) benchmarks.[1] Although bottle graphs are designed for multi-threaded applications, it is interesting to analyze the interaction between a single-threaded application and the Java virtual machine (JVM) threads. We use the default input set, unless mentioned otherwise.

For our experiments, we vary the number of application threads (2, 4 and 8 for the multi-threaded applications) and garbage collector threads (1, 2, 4 and 8). We experiment with different heap sizes (as multiples from the minimum size that each benchmark can run in), but we present results with two times the minimum to keep the heap size fairly small in order to exercise the garbage collector frequently so as to evaluate its time and parallelism components. We run the benchmarks for 15 iterations, and collect bottle graphs for every iteration individually, but present main results for the 13th iteration to show stable behavior.

We performed our experiments on an Intel Xeon E5-2650L server, consisting of 2 sockets, each with 8 cores, running a 64-bit 3.2.37 Linux kernel. Each socket has a 20 MB LLC, shared by the 8 cores. For our setup, we found that the number of concurrent threads rarely exceeds 8, with a maximum of 9 (due to a dynamic compilation thread). Therefore, we only use one socket in our experiments with HyperThreading enabled, which leads to 16 available hardware contexts. This setup

[1] Although eclipse spawns multiple threads, we found that the JavaIndexing thread is the only running thread for most of the time, so we categorize it as a single-threaded application.

| Benchmark | Suite | Version | ST/MT | Overhead |
|---|---|---|---|---|
| antlr | DaCapo | 2006 | ST | 0.40% |
| bloat | DaCapo | 2006 | ST | 0.64% |
| eclipse | DaCapo | 2006 | ST | 0.70% |
| fop | DaCapo | 2006 | ST | 0.20% |
| jython | DaCapo | 2009 | ST | 0.80% |
| luindex | DaCapo | 2009 | ST | 1.00% |
| avrora | DaCapo | 2009 | MT | 1.11% |
| lusearch | DaCapo | 2009 | MT | 0.32% |
| pmd | DaCapo | 2009 | MT | 0.44% |
| pseudoJBB | SPEC | 2005 | MT | 0.90% |
| sunflow | DaCapo | 2009 | MT | 0.03% |
| xalan | DaCapo | 2009 | MT | 0.91% |

**Table 5.1:** Considered benchmarks for bottle graphs and kernel module overhead. ST=single-threaded, MT=multi-threaded.

avoids data traversing socket boundaries, which could have an impact on performance that is hardware related. The availability of 16 hardware contexts does not trigger the OS to schedule out threads other than for synchronization or I/O.

We run all of our benchmarks on Jikes Research Virtual Machine version 3.12 [44]. We use the default best-performing garbage collector (GC) on Jikes, the stop-the-world parallel generational Immix collector [8]. In addition to evaluating the Jikes RVM, we also compare with the OpenJDK JVM version 1.6 [51]. We use their throughput-oriented parallel collector (also stop-the-world) in both the young and old generations. It should be noted that OpenJDK's compacting old generation has a different layout than Jikes' old generation, and thus will have a different impact on both the application and collector performance.

Because we use a stop-the-world collector, we can divide the execution of a benchmark into application and collection phases. Application and GC threads never run concurrently; therefore, the application and GC thread components in the bottle graph can be analyzed in isolation. For example, the total height of all application thread boxes is the total application running time, and the same holds for the GC threads. Also, because the collector never runs concurrently with other threads, the parallelism of the GC boxes is the parallelism of the collector itself.

## 5.4 Jikes RVM and Benchmark Analysis

In this section we will study the behavior of Java applications using bottle graphs. By varying the number of application threads, GC threads, heap size, and collecting results over many iterations, we have generated over 2,000 bottle graphs. We describe the main findings from this study in this section, together with a few bottle graphs that show interesting behavior. We refer the interested reader to the additional supporting material for all bottle graphs generated during this study, available at `http://users.elis.ugent.be/~kdubois/bottle_graphs_oopsla2013`.

We first define some terminology that will be used to describe the bottle graphs we gathered for Java. *Application work* is the sum of all active execution times of all application threads (excluding JVM threads), i.e., the total area of all application thread boxes.[2] Likewise, we define *application time* as the sum of all execution time shares of all application threads, i.e., the total height of the application thread boxes. Along the same lines, we define *garbage collection work* as the sum of all GC threads' active execution times, i.e., the total area of all GC thread boxes, and *garbage collection time* as the sum of all GC threads' execution time shares, i.e., the total height of the GC thread boxes.

We already showed and discussed bottle graphs for all our applications in Chapter 3, Section 3.3.2. We will now discuss collector and application performance and their impact on each other in Section 5.4.1 and 5.4.2. In Section 5.4.3, we also analyze the impact of the optimizing compiler by comparing the first and later iterations. We analyze why there is multi-threaded imbalance in one well-known benchmark, pmd, in Section 5.4.4. Finally we compare the performance of Jikes with OpenJDK in Section 5.4.5.

### 5.4.1 Garbage Collection Performance Analysis

We now explore what bottle graphs reveal about garbage collection performance, while varying the numbers of application and collection threads. We analyze in depth the variation in the amount of garbage

[2] We classify the MainThread as part of the application. The MainThread does some initialization and then calls the main method of the application. For single-threaded applications, all of the work is done in this MainThread. For multi-threaded applications, it does some initialization and then spawns the application threads.

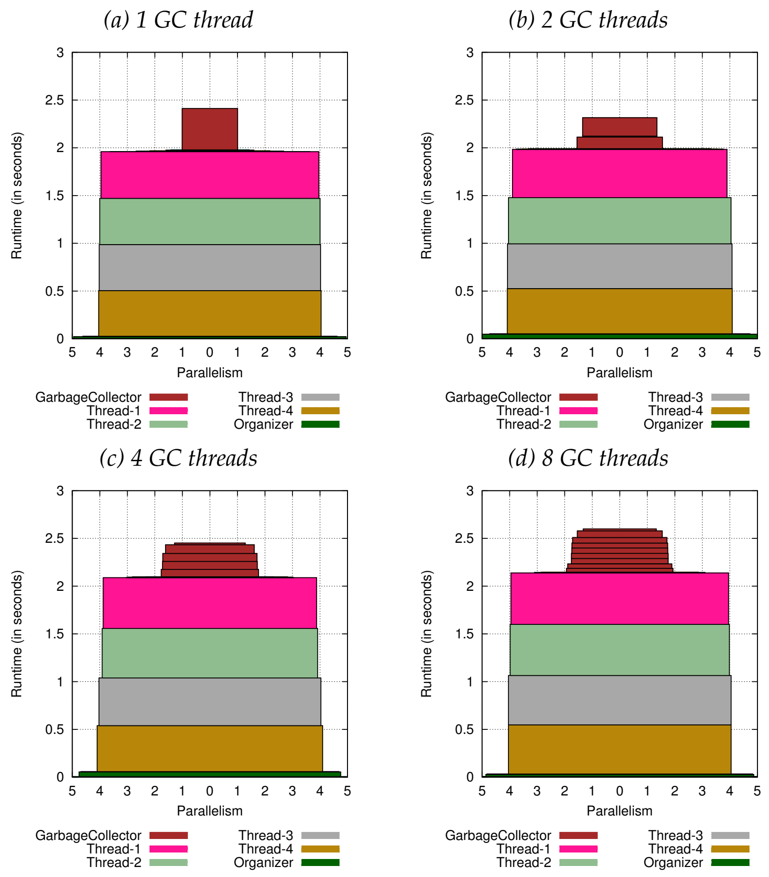


**Figure 5.1:** Xalan: scaling of GC threads with 4 application threads.

collection time (sum of GC box heights) and work (sum of GC box areas). Figures 5.1 and 5.2 show bottle graphs for xalan with increasing number of GC threads (Figure 5.1) and increasing number of application threads (Figure 5.2). We include only the xalan results here because this benchmark has representative behavior with respect to collection. Figure 5.3 shows the average collection work (a) and time (b) for all multi-threaded benchmarks, as a function of the number of GC threads and the number of application threads.[3] The values are normalized to the configuration with 2 application threads and 1 GC thread. Figure 5.4 shows the same data for the single-threaded benchmarks, obvi-

[3]We exclude the numbers for avrora and pseudoJBB for Figures 5.3, 5.5, 5.6, 5.7, 5.13, and 5.14. For these benchmarks, it is impossible to vary the number of application threads independently from the problem size.

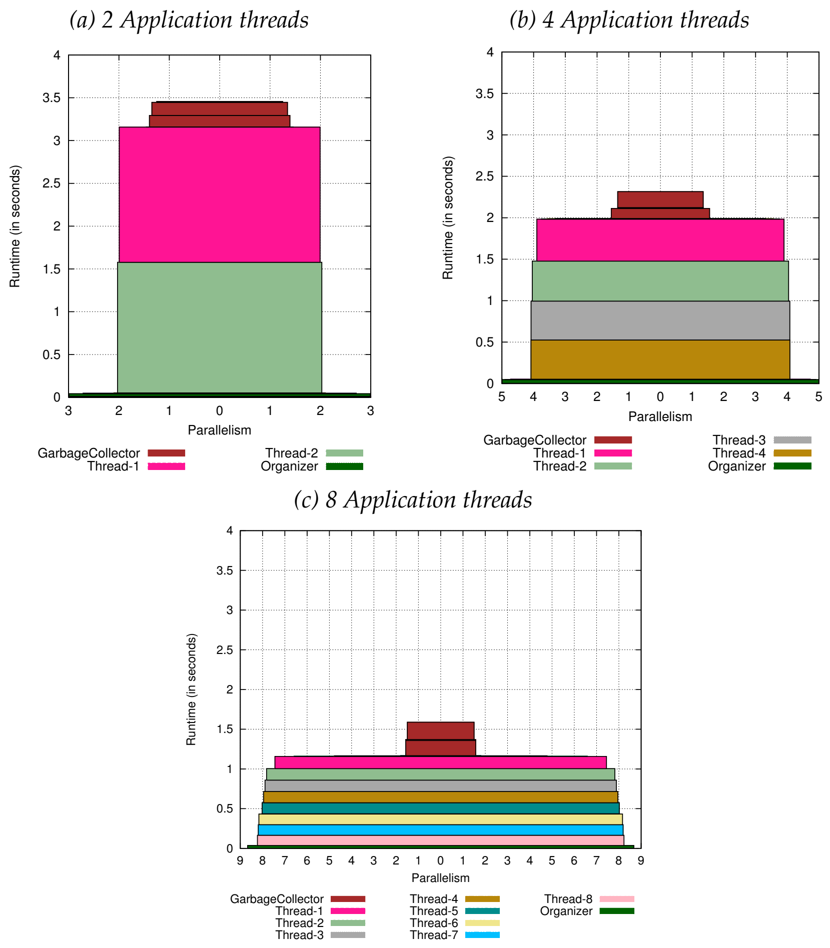
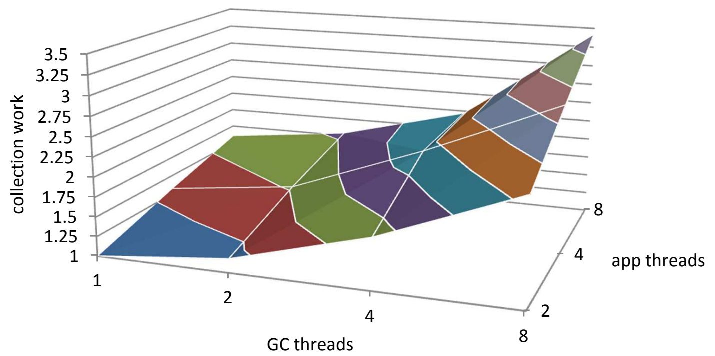


**Figure 5.2:** Xalan: scaling of application threads with 2 GC threads.

ously without the dimension of the number of application threads.

We make the following observations:

**Collection work increases with an increasing number of collection threads.** When the number of GC threads increases, the total collection work increases, i.e., the sum of the running times of all GC threads increases, see Figures 5.3 (a) and 5.4. For xalan, total collection work—which equals the total area of all GC thread boxes—increases by 73% from 1 to 8 GC threads (see Figure 5.1). For all multi-threaded benchmarks, there is an increase of 120% (averaged over all application *(a) Garbage collection work*



*(b) Garbage collection time*

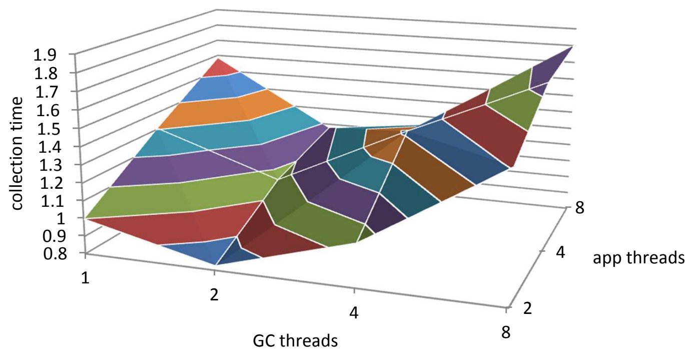


**Figure 5.3:** Average garbage collection work (a) and time (b) as a function of application and GC thread count (multi-threaded applications). Numbers are normalized to the 2 application and 1 GC thread configuration. *Collection work increases with increasing GC thread count and increasing application thread count, and 2 GC threads results in minimum collection time for all application thread counts.*

thread counts), and 198% for the single-threaded benchmarks.

There are a number of reasons for this behavior. First, more threads incur more synchronization overhead, i.e., extra code to perform synchronization due to GC threads accessing shared data. Figure 5.14 in Section 5.4.5 (Jikes line) shows that the number of futex calls significantly increases as the number of GC threads increases. Second, because garbage collection is a very data-intensive process and the last-level cache (LLC) is shared by all cores, more GC threads will lead to more interference misses in the LLC, leading to a larger running time. Figure 5.5 shows the number of LLC cache misses as a function of

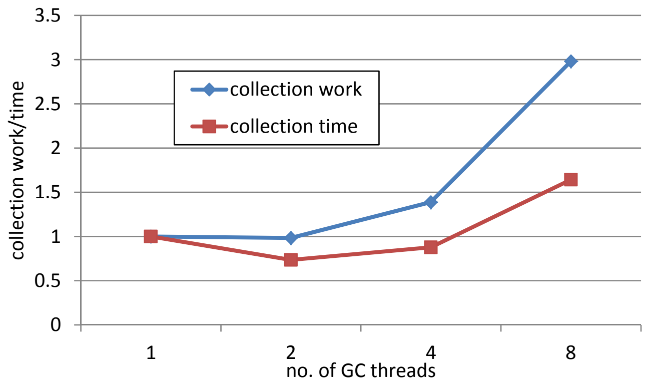


**Figure 5.4:** Average collection work and time as a function of GC thread count (single-threaded applications). Numbers are normalized to the 1 GC thread configuration. *Collection work increases with increasing GC thread count and collection time is minimal with 2 GC threads.*

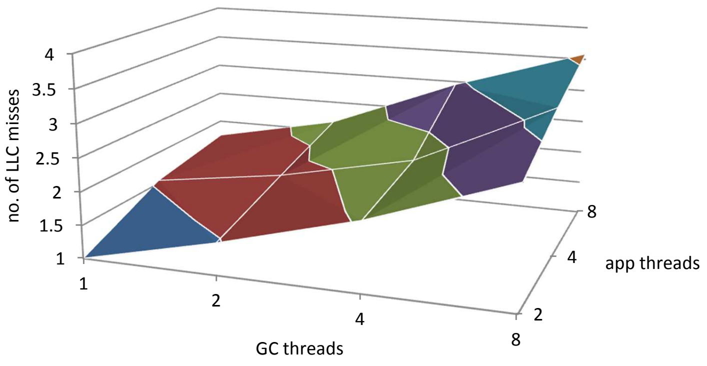


**Figure 5.5:** Average number of LLC misses as a function of application and GC thread count (multi-threaded applications). Numbers are normalized to the 2 application and 1 GC thread configuration.

the number of GC threads and the number of application threads for all multi-threaded applications, normalized to the 2 application, 1 GC thread configuration. The number of LLC cache misses increases significantly when the number of GC threads increases (more than 2.5 times for 8 GC threads compared to 1 GC thread), which raises the total execution time of the GC threads.

**Collection work increases with an increasing number of application threads.** There is an increase in the amount of time garbage collection

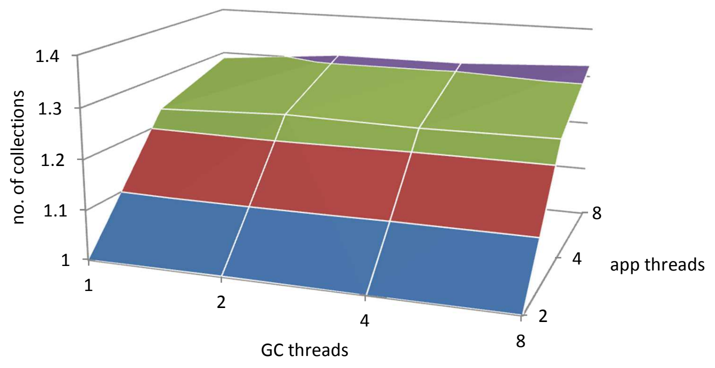


**Figure 5.6:** Average number of collections as a function of application and GC thread count (multi-threaded applications). Numbers are normalized to the 2 application and 1 GC thread configuration.

threads are actively running (or their total work), as we go to larger application thread counts. For xalan, collection work increases by 60% if the number of application threads is increased from 2 to 8 (see Figure 5.2). For all multi-threaded benchmarks, we see an average increase of 47% (over all GC thread counts), in Figure 5.3 (a). To explain this behavior, we refer to Figure 5.6, which shows the average number of collections that occurred during the execution of the multi-threaded benchmarks,[4] again as a function of the number of GC and application threads. When increasing the number of GC threads, the number of collections does not increase. However, the number of application threads has a clear impact on the number of collections: the more application threads, the more collections. The intuition behind this observation is that more threads generate more live data in a fixed amount of time (because Jikes has thread-local allocation where every thread gets its own chunk of memory to allocate into), so the heap fills up faster compared to having fewer application threads. The more collections, the more collection work that needs to be done.

**2 GC threads are optimal regardless of the number of application threads.** Figures 5.3 (b) and 5.4 show that garbage collection time is minimal for two GC threads in Jikes, even for lower and higher appli-

[4]We exclude pmd from this graph, because due to its imbalance, only one thread is running during a significant portion of its execution time. Since this portion increases with the number of application threads, the number of collections decreases, while we see an increase for all other benchmarks.

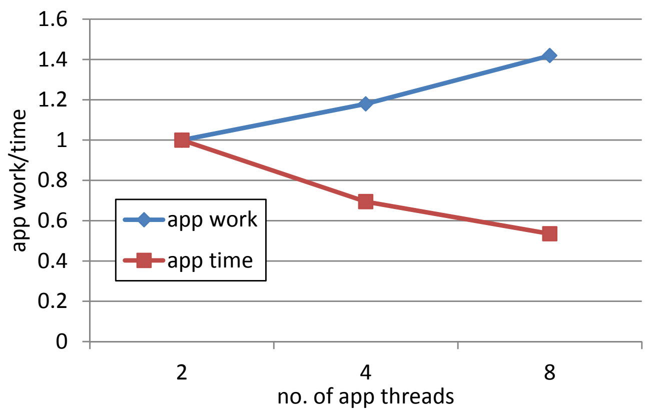


**Figure 5.7:** Average application work and time as a function of application thread count (multi-threaded applications). Numbers are normalized to the 2 application thread configuration. *Application time decreases with increasing application thread count, but the decrease is limited because application work increases with more application threads.*

cation thread counts. Although the amount of work increases when the number of GC threads is increased from 1 to 2, collector parallelism is also increased (from 1 to 1.5), leading to a lower net collection time. When the number of GC threads is further increased to 4 and 8, parallelism increases only slightly (to 1.7 and 1.8, respectively), and does not compensate for the extra amount of work, leading to a net increase in collection time. Although we present results here for only two times the minimum heap size, we found two GC threads to be optimal for other heap sizes as well.

## 5.4.2 Application Performance Analysis

We analyze changes in the application time and work as we vary the number of application threads, which seem to suffer less from limited parallelism than the garbage collector. Figure 5.7 shows the application execution time (excluding garbage collection time) and work as a function of the number of application threads (for the multi-threaded applications), keeping the GC threads at two. Numbers are normalized to those for two application threads.

**Application time decreases with an increasing number of application threads, but the decrease is limited because application work**

**increases due to the overheads of parallelism.** When the application thread count is increased, application time does decrease, but not proportionally to the number of threads. Compared to two application threads, execution time decreases with a factor of 1.4 for four application threads and 1.9 for 8 threads. Part of the reason this decrease is not higher is the limited parallelism (average application parallelism equals 1.9, 3.1 and 4.7, for 2, 4 and 8 application threads, respectively). However, this does not fully cover the smaller application speedup: from 2 to 8 application threads, parallelism is increased by a factor of 2.5, while execution time is reduced by a factor of only 1.9. This difference is explained by the fact that application work also increases with the number of application threads, see Figure 5.7. This increase is due to synchronization overhead (the number of futex calls for the application increases from 1.5 to 4.7 calls per ms when the number of application threads increases from 2 to 8) and an increasing number of LLC misses due to more interference (see also Figure 5.5). Increasing the number of application threads leads to more application work and increased parallelism, resulting in a net reduced execution time, but due to the extra overhead, the execution time reduction is smaller than the thread count increase.

### 5.4.3 Compiler Performance Analysis

Lastly, while generating our bottle graphs across many benchmark iterations, we noticed the difference between startup and steady state behavior discussed in detail in Java methodology research [9]. Figure 5.8 shows the bottle graphs of one single-threaded benchmark, jython, during the first, 9th and 11th iterations. During the first iteration, we see a large overall execution time, and a large (one second) time share for the Organizer thread. This JVM service thread performs dynamic compilation concurrently with the application (it has a parallelism of two), and thus is very active in the first iteration, but is much more minimal in iteration nine. Iteration nine has a reduced execution time because the Java source code is now optimized and the benchmark is running more at steady-state. However, the bottle graph for iteration 11 shows an increased Organizer thread component. This behavior is specific to this benchmark; other applications see the Organizer box disappear in all higher iterations. Jython is different in that it dynamically interprets python source code into bytecode, and then runs it. Jython actually runs a benchmark within itself, and Jikes continues to optimize the gen-

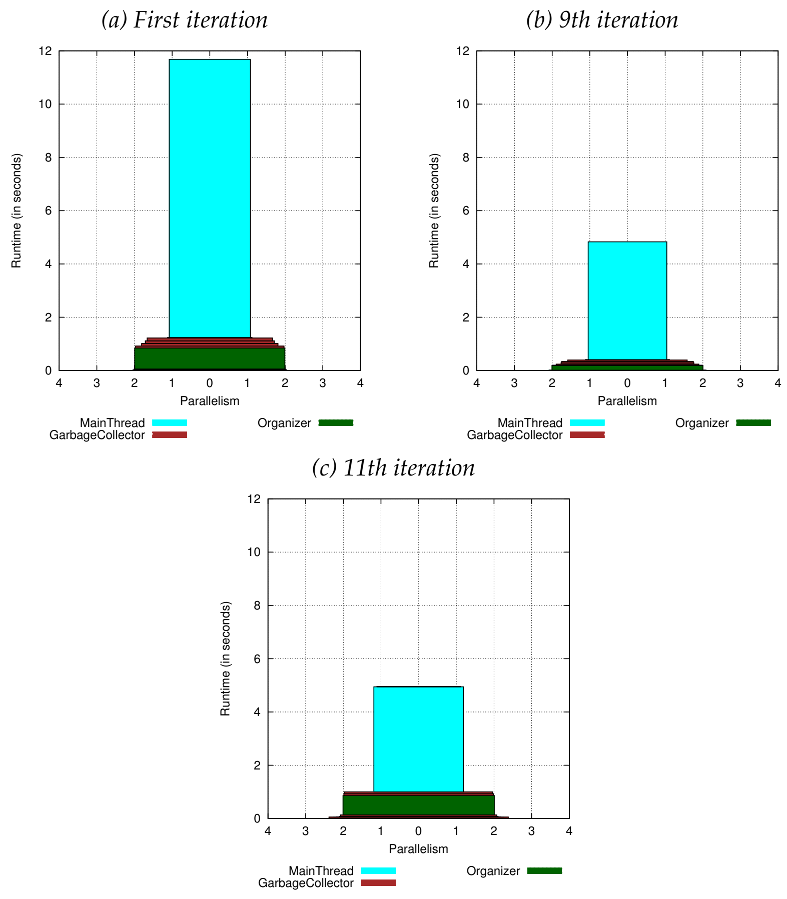


**Figure 5.8:** Jython: behavior of Organizer thread over different iterations for 4 GC threads.

erated bytecode with the optimizing compiler at various iterations, because the compiler is probabilistic and is triggered unpredictably. Thus, bottle graphs are also useful for seeing program and JVM variations between iterations.

### 5.4.4 Solving the Poor Scaling of Pmd

We have analyzed the performance of both Java applications and Jikes RVM's service threads using bottle graphs. As shown in Figure 3.6, Chapter 3, pmd has one thread that has significantly limited parallelism. We now analyze this bottleneck and propose suggestions on

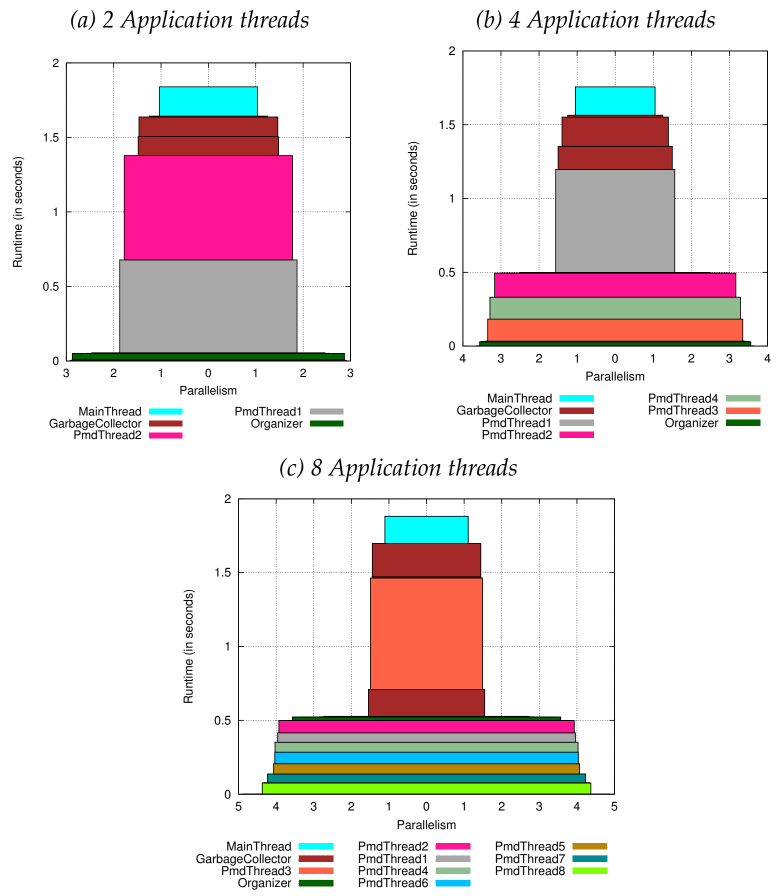


**Figure 5.9:** Pmd: scaling of application threads with 2 GC threads (default input set).

how to fix pmd's scalability problem.

Figure 5.9 shows the bottle graphs for pmd for 2, 4 and 8 application threads, while keeping the collection threads at two and using the default input set. With two application threads, the left graph shows these threads have approximately the same height and width (a parallelism close to two). However, for 4 and 8 threads, pmd clearly has an imbalance issue: there is one thread that has less parallelism and a larger execution time share than the other threads. To understand the cause, we gathered bottle graphs at several time intervals (every 0.5 seconds) within the 13th iteration of the benchmark, running with 2 GC and 8 application threads, shown in Figure 5.10. We see that after

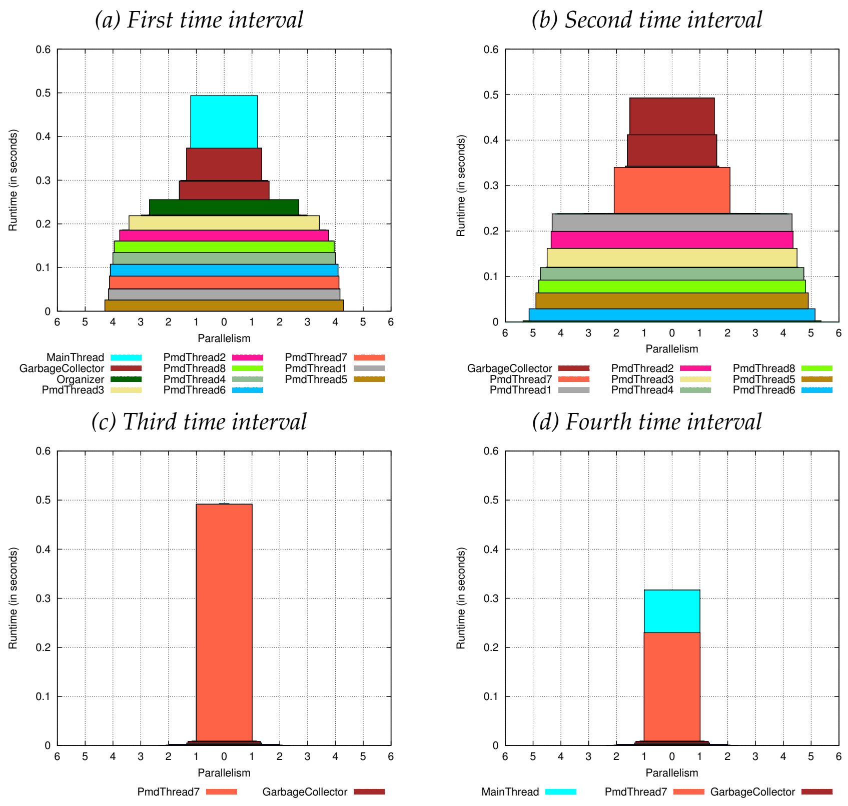


**Figure 5.10:** Pmd: bottle graphs taken every 0.5 seconds with 2 GC threads and 8 application threads (default input set).

the first 0.5 seconds, although there is some variation, the application threads are still fairly balanced in regards to parallelism. Starting from the second time interval, one application thread has limited parallelism and a larger share of execution time (PmdThread7). After one second of execution, that same thread continues to run alone while all other application threads have finished their work.

Pmd is a (Java) source code analyzer, and finds unused variables, empty catch blocks, unnecessary object creation, etc. It takes as input a list of source code files to be processed. The default DaCapo input for pmd is a part of the source code files of pmd itself. There is also a large input set, which analyzes all of the pmd source code files. Internally, pmd is parallelized using a work stealing approach. All files are put in a queue, and the threads pick the next unprocessed file from the queue when they finish processing a file. Compared to static work partition-

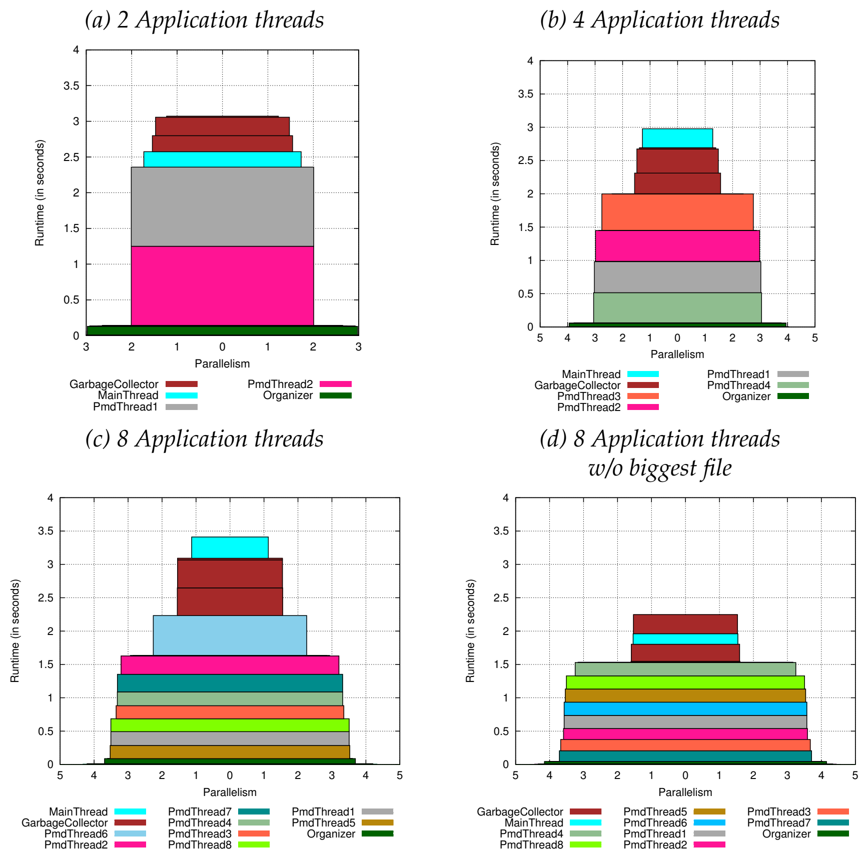


**Figure 5.11:** Pmd: scaling of application threads with 2 GC threads (large input set). For the fourth graph, the biggest source file is removed from the input set.

ing, work stealing normally improves balance, because one thread can process a large job, while another thread processes many small jobs. Imbalance can only occur when one or a few jobs have such a large process time that there are not enough other small jobs to be run in parallel. This is exactly the case for the default input set of pmd. There are 220 files, with an average size of 3.8 KB. However, there is one file that is 240 KB, which is around 63 times larger than the average. Therefore, the thread that picks that file will always have a larger execution time than the other threads, and there are not enough other files to keep the other threads concurrently busy.

This problem is partly solved when using the large input set, which also has the same big file, but there are 570 files in total, so more files can be processed concurrently with the big file. Figures 5.11(a)–(c) show the bottle graphs for the large input set with 2, 4 and 8 application threads. The imbalance problem is solved for 2 and 4 application threads, but is still present for 8 threads (although less pronounced compared to the default input set). The more threads there are, the more other jobs are needed to run concurrently with the large job. Figure 5.11(d) shows the bottle graph for 8 threads and the large input set excluding that one big file, which leads to a balanced execution.

We can conclude that users of pmd should make sure that there is no file that is much larger than the others to prevent an imbalanced, and therefore inefficient, execution. The balance can also be improved by making the scheduler in pmd more intelligent. For example, the files to be processed can be ordered by decreasing file size, such that big files are processed first and not after a bunch of smaller files. In that case, there are more small files left for the other threads, and balance is improved. Another, probably more intrusive, solution is to provide the ability to split a single file across multiple threads.

Apart from imbalance, there is also a problem of limited parallelism in pmd. Figure 5.11(d) shows that the parallelism of 8 application threads is only 3.5. We looked into the code and found a synchronized map data structure that is shared between threads and guarded by one lock. Reducing the time the lock is held and/or using fine-grained locking should improve parallelism, and therefore performance, for pmd.

### 5.4.5 Comparing Jikes to OpenJDK

In Section 5.4.1 and 5.4.2 we analyzed the performance of the garbage collector and the application with Jikes RVM. We made two novel observations: collector parallelism is limited, leading to an optimal GC thread count of 2, and the number of GC threads and the number of application threads have an impact on the amount of collection work. We present here a similar analysis on the OpenJDK virtual machine, revealing that OpenJDK's garbage collector scales better than in Jikes, benefiting from up to 8 GC threads. However, we find that collection work still increases with the number of application threads and GC threads.

Figure 5.12 shows the bottle graphs for pseudoJBB with 4 appli-

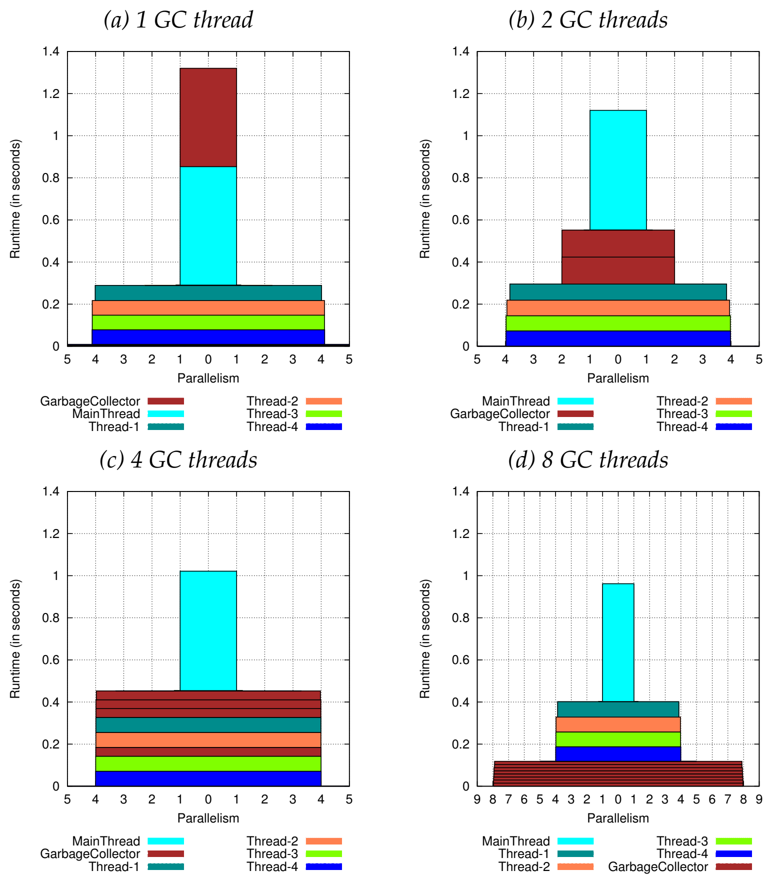
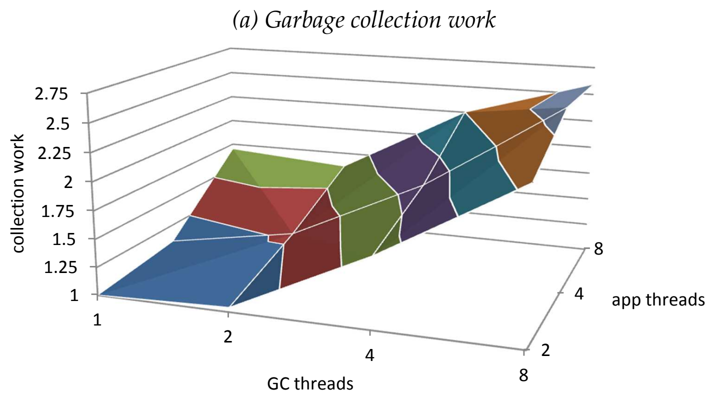


**Figure 5.12:** PseudoJBB: scaling of GC threads on OpenJDK, with 4 application threads.

cation threads on OpenJDK, with an increasing GC thread count. We chose pseudoJBB graphs here because they are most illustrative of collector behavior, other benchmarks have similar behavior. It is immediately clear that GC scales much better for OpenJDK than for Jikes. The average collection parallelism across all multi-threaded benchmarks is 1, 1.9, 3.3 and 4.5, for 1, 2, 4 and 8 GC threads, respectively. This parallelism is substantially larger than the 1.8 parallelism for 8 GC threads on Jikes.

We further investigate the performance of OpenJDK by viewing its collection work and time as a function of GC and application thread count in Figure 5.13. We present average collection work (a) and time (b) for the multi-threaded benchmarks normalized to the 1 GC and 2



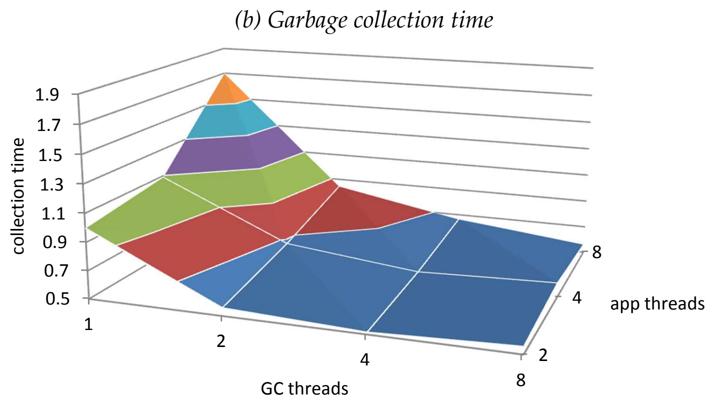


**Figure 5.13:** Average OpenJDK garbage collection work (a) and time (b) as a function of application and GC thread count (multi-threaded applications). Numbers are normalized to the 2 application and 1 GC thread configuration. *Garbage collection on OpenJDK scales better than on Jikes, but collection work also increases with increasing GC thread count and increasing application thread count.*

application thread configuration (contrasted with Figure 5.3 for Jikes). We confirm that OpenJDK scales better than Jikes by seeing that collection time decreases with an increasing number of GC threads in Figure 5.13(b). There is less synchronization during garbage collection in OpenJDK compared to Jikes, as evident in Figure 5.14 which shows the number of futex calls per unit of time as a function of the number of GC threads. Although the number of futex calls also increases with increasing GC thread count, the increase is much smaller for OpenJDK than for Jikes.

For OpenJDK, we found 4 GC threads to be optimal with either 2 or 4 applications threads, and 8 GC threads optimal for 8 applica-

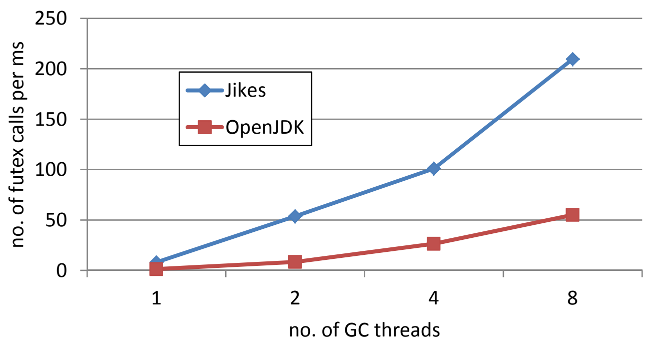


**Figure 5.14:** Average number of futex calls per ms during garbage collection as a function of GC thread count, with 4 application threads (multi-threaded applications).

tion threads. Garbage collection on OpenJDK scales better than for Jikes RVM, but we observe that collector time slightly increases or does not decrease much between 4 and 8 GC threads, which suggests that garbage collection scaling on OpenJDK saturates at 4 to 8 GC threads. This is in line with the findings in [34], where the authors observe a decrease in collection time when the number of GC threads is increased from 1 to 6, but an increase when the number of GC threads is increased to 12 and more.

Our two other observations for Jikes, namely that collection work increases with GC thread count and application thread count, still hold for OpenJDK. Figure 5.13(a) for OpenJDK looks very similar to Figure 5.3(a) for Jikes. While we have observed in both JVMs a saturation point in the utility of increasing the parallelism of garbage collection threads, we still find that inter-thread synchronization has a significant impact on garbage collection performance.

## 5.5 Related Work

We now describe related work in Java performance analysis. We detail one particularly related visualization tool called WAIT in Section 5.5.1.

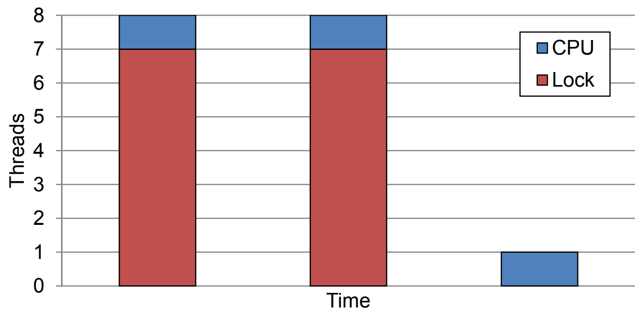


**Figure 5.15:** Output of WAIT for pmd running on OpenJDK with 8 application and 2 GC threads using a 1 second sampling rate. The graph shows 3 samples that monitor application thread status: *CPU* threads are active, while *Lock* threads are inactive.

### 5.5.1 Comparison to IBM WAIT

IBM WAIT[5] [1] is a performance visualization tool for diagnosing performance and scalability bottlenecks in Java programs, particularly server workloads. It uses a light-weight profiler that collects samples of information about each thread at regular points in time (configurable through a sampling rate parameter). WAIT records information on each thread's status (active, waiting or idle), locks (held or waiting for), and where in the code the thread is executing. This data is used to construct a graph that visualizes the threads' status over time (x-axis), each bar showing a sample point. The bar's height is the total number of threads for that sample, and the bar is color-coded by thread status. Figure 5.15 shows the output of WAIT for pmd running on OpenJDK with 8 application threads and 2 GC threads during the 13th iteration using a 1 second sampling rate, where *CPU* denotes active threads, and *Lock* denotes inactive threads. Information about code position and locks can be retrieved by clicking on the bars in the timeline.

While WAIT is a powerful analysis tool for Java programs, it has some limitations. First, it can be applied only to Java application threads, not to parallel programs written in other languages or to Java virtual machine service threads, both of which can be analyzed easily with our bottle graphs because we use OS modules. Second, WAIT is sampling-based, and thus collects a snapshot of information only at specific program points, with increasing overhead with finer-

[5]`https://wait.ibm.com/`

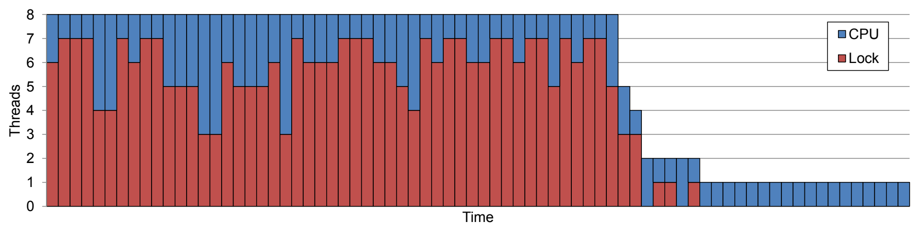


**Figure 5.16:** Output of WAIT for pmd running on OpenJDK with 8 application and 2 GC threads using a 50 millisecond sampling rate. The graph shows one bar per sample that monitors application thread status: *CPU* threads are active, while *Lock* threads are inactive.

granularity sampling. To demonstrate this, Figure 5.15 shows results for pmd using at the lowest default sampling period value of 1 second. WAIT only collects 3 samples, which suggest that there is only one active (*CPU*) thread during the execution. We lowered the sampling rate to 50 milliseconds by modifying WAIT's scripts, and produced the more detailed graph in Figure 5.16 which reveals that the number of active threads varies over time. However, the overhead of using the 1 second versus 50 millisecond sampling period jumps from 0.87% to 16.62%, an order of magnitude larger than for our tool.

In contrast, bottle graphs contain much more information for lower overhead. Our OS modules are continually monitoring *every* thread status change, and *aggregating our execution time share and parallelism metrics* at all times in a multi-threaded program run, on a *per-thread basis*. For roughly the same overhead, we contrast Figure 5.10 showing bottle graphs at various times in a run of pmd with Figure 5.15 showing WAIT's three thread samples. WAIT's visual representation makes it hard to know that one of pmd's threads is a bottleneck throughout the run. In the end, pmd's parallel imbalance due to input imbalance is difficult to detect without analyzing the source code and input. In conclusion, bottle graphs' visualization of scalability per-thread facilitates grouping by category (such as for thread pools or garbage collection threads) in order to analyze the group's execution time share, parallelism, or work to pinpoint parallelism imbalances.

### 5.5.2 Java Parallelism Analysis

Analyzing Java performance and parallelism has become an active area of research recently. Most of these studies use custom-built analyzers to measure specific characteristics of interest. For example, Kalibera et al. [36] analyze concurrency, memory sharing and synchronization behavior of the DaCapo benchmark suite. They provide concurrency metrics and analyze the applications in-depth, focusing on inherent application characteristics. They do not provide a visual analysis tool to measure and quantify performance and scalability, and reveal bottlenecks on real hardware as we do.

Researchers recently analyzed the scalability problems of the garbage collector in the OpenJDK JVM [34]. They also confirm that the total collection times increase with the number of collector threads, without providing a visualization tool. They did follow-on work to optimize scalability at large thread-counts for the parallel stop-the-world garbage collection in OpenJDK [29]. Similarly, Chen et al. [11] analyzed scalability issues in the OpenJDK JVM, and provided explanations at the hardware level by measuring cache misses, DTLB misses, pipeline misses, and cache-to-cache transfers. They also explored the sizing of the young generation and measured the benefits of thread-local allocation. They did not, however, vary the number of collection threads or explore the scalability limitations of the parallel collector itself, or how it interacts with the application, as we do in this work.

## 5.6 Summary

In this chapter we used bottle graphs to analyze Java applications. Bottle graphs are an intuitive and useful tool for visualizing per-thread performance of a multi-threaded application. We showed how bottle graphs can be constructed using light-weight OS modules, allowing us to generate bottle graphs of unmodified applications running on native hardware. We then illustrated the usefulness of bottle graphs by doing an experimental study of 12 Java benchmarks running on top of Jikes RVM. We studied the scaling of application and JVM service threads, revealing scalability limitations in several well-known applications (avrora and pmd), and poor scaling of the garbage collection threads on Jikes RVM. We compared this poor scaling of the garbage collector against OpenJDK's garbage collector, which apparently scales much better for our thread counts.

Chapter 6

# Speedup Stacks: Analyzing Application Scaling

*In this chapter we study how the applications as a whole scale by using our third performance analysis method, speedup stacks.*

## 6.1 Introduction

Speedup stacks are meant for analyzing the scalability of an entire application, by quantifying the various causes of sub-linear speedup. This provides insight to programmers on why their multi-threaded programs' speedup over its single-threaded version is not actually proportional to the number of cores and threads. In this dissertation we present two versions of speedup stacks, a first version that uses hardware support for measuring speedup delimiters, and a second version that uses software support and targets managed language applications.

We first describe a method for computing a speedup stack from a single multi-threaded execution of an application, by making use of an additional hardware component. We validate the accuracy of this approach across a set of SPLASH-2, PARSEC and Rodinia benchmarks. We also describe several applications for speedup stacks apart from the obvious application of analyzing performance scaling bottlenecks. We use speedup stacks to classify benchmarks based on their scaling bottlenecks, we identify optimization opportunities, and we analyze LLC performance.

We then show how speedup stacks for managed language applications can be generated using a profiling method that is implemented in software. We use this method for analyzing scaling behavior of Java applications running on Jikes RVM with both a stop-the-world and concurrent garbage collector.

## 6.2 Speedup Stacks Measured in Hardware

As explained in the Chapter 3, in order to compute a speedup stack, we need to break up multi-threaded execution time into its various cycle components. In this version of speedup stacks we include the following speedup delimiters for multi-threaded workloads on multi-core hardware: resource sharing (which can either have a positive or a negative impact on performance), synchronization, cache coherence, load imbalance and parallelization overhead.

### 6.2.1 Constructing Speedup Stacks

For measuring these scaling delimiters we developed a tool that uses additional hardware support. The reason for choosing for hardware support is because it is difficult to measure the impact of resource sharing on performance precisely with a software only solution. Therefore, we designed a dedicated counter architecture in hardware that measures the interference between threads running on a multi-core processor. The next sections describe this counter architecture and how it measures both negative and positive interference between threads, subsequent sections then describe how we measure the other scaling delimiters needed for this version of speedup stacks.

#### Resource Sharing

Our counter architecture makes a distinction between two sources of negative inter-thread interference due to resource sharing: (i) inter-thread misses in the shared cache (i.e., the per-thread miss rate increases due to conflicts induced by other threads), and (ii) resource and bandwidth contention in the memory subsystem which causes intra-thread misses to take longer — we will refer to these additional cycles as *waiting cycles*. We first discuss how the counter architecture measures those two types of negative interference, we then discuss how the counter architecture accounts for positive interference between threads.

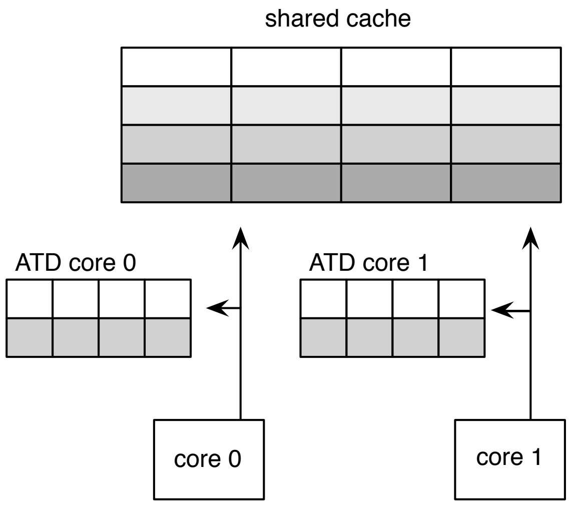


**Figure 6.1:** The ATD samples a number of sets in the shared cache to identify inter-thread misses.

**Inter-thread misses.** We detect inter-thread misses using a structure called the Auxiliary Tag Directory (ATD). The inter-thread miss performance impact is then estimated through an accounting mechanism.

The Auxiliary Tag Directory (ATD) is a structure private to each core that keeps track of what the status of the shared cache would be if it were private to that core, see also Figure 6.1. The ATD was first proposed by Qureshi and Patt [52] to keep track of the utility of the various ways in a shared cache to each of the cores; we use the ATD for a different purpose. The ATD keeps track of the tags and replacement bits (not the data) due to accesses by the given core. An access to the shared cache accesses both the shared cache and the private ATD of that core. Only the shared cache returns data (from the cache itself if the access results in a hit, or from main memory if it is a miss), but both the shared cache and the ATD adjust the tag and replacement bits. An LLC miss is classified as an intra-thread miss if it also misses in the ATD; in case of a hit in the ATD, the LLC miss is classified as an inter-thread miss.

Now that we know which misses in the shared cache are inter-thread versus intra-thread misses, the next question is to determine what their performance penalty is. We measure the number of interference cycles (miss penalty) due to inter-thread misses as the number of cycles an inter-thread miss blocks the head of a full ROB (an insight provided by interval analysis [25]). The mechanism is as follows: as soon as the ROB is full and an inter-thread miss is at the ROB head, we start counting interference cycles. This requires being able to detect that the ROB is full, and a bit per ROB entry to keep track of whether a load miss is an inter-thread miss.

In spite of the fact that the ATD contains only tag and replacement bits and no data, the hardware overhead is substantial: more than 6% of a 2MB shared cache *per core*, and thus the overhead for a multi-core processor with a large number of cores quickly becomes substantial and practically infeasible. We therefore use set sampling [38] to reduce the hardware overhead. We evaluated different sampling rates and found that sampling 32 out of 4096 sets yields the best balance between hardware overhead and accuracy. The hardware overhead of the sampled ATD equals less than 0.05% of the shared cache per core. However, because of set sampling we are unable to detect all inter-thread misses; we only know the status (inter-thread versus intra-thread miss) for those accesses that are sampled in the ATD. We therefore have to estimate the total number of interference cycles. We considered two approaches.

The first approach, the extrapolation approach, measures the penalty of the sampled inter-thread misses only, and then extrapolates to all inter-thread misses by multiplying by the sampling ratio.

$$\begin{aligned}\textit{inter-thread miss penalty} &\approx \textit{sampled inter-thread miss penalty}\\ &\quad\times\frac{\textit{no. of cache accesses}}{\textit{no. of sampled cache accesses}}.\end{aligned} \tag{6.1}$$

The second approach, the interpolation approach, measures the penalty of all LLC misses (both inter-thread and intra-thread misses), and estimates the fraction of this penalty due to inter-thread misses by taking the ratio of the number of sampled inter-thread misses to the total number of sampled misses:

$$\begin{aligned}\textit{inter-thread miss penalty} &\approx \textit{total miss penalty}\\ &\quad\times\frac{\textit{no. of sampled inter-thread misses}}{\textit{no. of sampled misses}}.\end{aligned} \tag{6.2}$$

We compared the accuracy of both approaches and the conclusion is that the extrapolation approach is slightly more accurate. The reason is that the extrapolation approach measures the penalties of the sampled inter-thread misses, whereas the interpolation approach calculates the penalty for all misses (both inter-thread and intra-thread misses). The penalty for all misses may not be representative for the inter-thread misses, hence the extrapolation approach tends to be more accurate. We will therefore use the extrapolation approach in our measurement tool.

**Waiting cycles for intra-thread misses.** Intra-thread misses can have additional waiting cycles that prolong the latency seen for these misses due to resource and bandwidth contention in the memory subsystem: memory bus conflicts, bank conflicts and inter-thread row buffer misses. We now discuss how the counter architecture counts the number of waiting cycles and how to estimate their impact on overall performance.

*Bus contention.* When a memory operation from one core occupies one of the buses (command, address or data bus) while a memory operation from another core also wants to access the bus, then the latter incurs waiting cycles that would not have occurred in isolated execution. This is detected by inspecting the bus owner if a memory operation is ready to be scheduled on the bus and the bus is occupied. If the bus is owned by another core, then waiting cycles are accounted for.

*Bank contention* An access that is delayed due to its destination bank being occupied by an access of another core, is detected similarly, and the extra waiting cycles is accounted for.

*Inter-thread row buffer misses* This type of interference occurs when a row buffer hit in isolated execution becomes a row buffer miss during multi-core execution, in case of an open-page memory policy. This occurs when a memory operation of another core accesses another row between the two consecutive accesses of one core to the same row. Since row buffer misses take considerably more time to be serviced than row buffer hits, this introduces an additional penalty and should be accounted for as waiting cycles.

Inter-thread row buffer misses are detected by maintaining an Open Row Array (ORA) per core (e.g., in the memory controller), see Figure 6.2. The ORA keeps track of the ID of the most recently accessed row per memory bank per core. If a row buffer miss hits in the private ORA, then the miss is caused by interference, and the extra penalty (i.e., the difference between the closed and open page access times) is accounted for as waiting cycles. In case of a closed-page policy, the ORA is not needed.

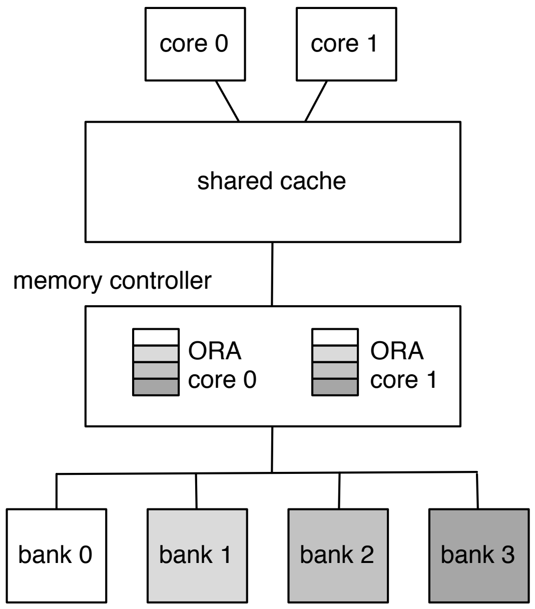


**Figure 6.2:** The ORA keeps track of the most recently accessed row per memory bank per core.

*Hardware prefetching* A long-latency load miss that also appears in the hardware prefetch queue (i.e., the prefetch is to be issued and/or is underway), is accounted waiting cycles if the load blocks commit at the head of a full ROB. This strategy basically assumes that the prefetch would be timely in isolated execution — while it is not during multi-core execution because of contention. Although this is not always the case, this assumption keeps the accounting architecture simple and we found it to account for a major fraction of the interference due to prefetching.

The waiting cycles need to be kept track of for each individual memory access because there may be multiple outstanding inter-thread misses (whose latency is potentially hidden). The counter architecture keeps track of the waiting cycles in the MSHRs: we add a waiting cycle counter (10 bits) to each MSHR entry, and we add all waiting cycles pertaining to this memory access to this counter.

To estimate the performance impact of the waiting cycles, we again use the insights provided by interval analysis [25]: a long-latency load miss only has impact on overall performance if it blocks the head of the ROB and causes the ROB to fill up. This implies that waiting cycles need to be accounted as interference cycles only if the long-latency load miss makes it to the ROB head and fills up the ROB. Based on this insight, we propose the following mechanism: if a miss blocks the head of the ROB and causes the ROB to fill up, then we add the miss' waiting cycles (that are kept track of in the MSHRs) to the per-core interference cycle counter.

We need to account for waiting cycles for the intra-thread misses only — the additional penalty incurred by inter-thread misses is accounted for as described before. The number of intra-thread misses is not readily available though (because of set sampling in the ATD). Therefore we need to extrapolate on the sampled sets in the ATD. In line with what we described above, we again consider two approaches.

The extrapolation approach measures the waiting cycles of the sampled intra-thread misses and then extrapolates by multiplying with the ratio of cache accesses versus the number of sampled accesses:

$$\begin{aligned} \textit{total waiting cycles} &\approx \textit{waiting cycles sampled intra-thread misses} \\ &\quad \times \frac{\textit{no. of cache accesses}}{\textit{no. of sampled cache accesses}}. \end{aligned} \tag{6.3}$$

The interpolation approach takes the number of waiting cycles of all misses and multiplies that with the estimated fraction of intra-thread misses. This approximates the number of waiting cycles for all intra-thread misses.

$$\begin{aligned} \textit{total waiting cycles} &\approx \textit{waiting cycles for all misses} \\ &\quad \times \frac{\textit{no. of sampled intra-thread misses}}{\textit{no. of sampled misses}}. \end{aligned} \tag{6.4}$$

For the same reasons as the ones discussed earlier, we find that extrapolation is more accurate than interpolation. Therefore, we also use the extrapolation approach to measure the total number of waiting cycles in our measurement tool.

**Positive interference.** Besides negative interference, threads of a multi-threaded application also exhibit positive interference as threads share data, which implies that one thread can load data into the shared LLC that can possibly be reused by other threads. This means that the other threads will experience a hit instead of a miss in case of a private

LLC. This sharing effect thus has a positive impact on performance and should be measured by the cycle accounting architecture.

We refer to an LLC hit as an *inter-thread hit* if the thread accesses data that was previously brought into the shared LLC by another thread. An inter-thread hit can be detected by our counting architecture using the aforementioned ATDs: a hit in the shared LLC that results in a miss in the private ATD, is classified as an inter-thread hit.

To quantify the impact of inter-thread hits on performance, we need to estimate the penalty an access would have seen if it were a miss. We cannot use an extrapolation technique here, because there is no penalty, hence we cannot measure it. Instead, we use an interpolation approach: we take the total number of cycles a core is stalled on an LLC load miss, and divide that by the number of LLC load misses. This yields the average miss penalty. Because we use sampling in the ATDs we do not know the exact number of inter-thread hits. To estimate this number we take the number of sampled inter-thread hits and multiply it with the sampling factor (total number of LLC accesses divided by the number of sampled ATD accesses). We then multiply this number with the average miss penalty to obtain an estimate for the total positive interference.

### Spinning

After resource sharing, spinning is the second scaling delimiter we discuss. Spinning happens when a thread wants to acquire a lock to enter a critical section, but the lock is in use by another thread. Similarly, spinning may happen on a barrier. In that case, the thread enters a spin loop and constantly checks the lock until it is released. This implies that the time a thread spends in a spinning loop should be accounted as interference cycles.

To detect spinning and account for its interference cycles, we implemented and evaluated two spinning detection mechanisms that have been proposed in literature. Li et al. [43] propose a mechanism where all backward branches are monitored and considered as possible spinning loop branches. If the processor state is unchanged since the last occurrence of the same branch, then the loop is considered a spinning loop. Processor state is tracked using a compact representation to represent register state changes, and when a (non-silent) store occurs, processor state is assumed altered. By keeping a timestamp at the occurrence of backward branches, and subtracting this timestamp from the current time (when the same branch is executed and processor state is unchanged), one can quantify the time spent in spin loops.

A second mechanism, proposed by Tian et al. [60], detects spinning by monitoring loads, as a spin loop contains at least one load (to check the lock status). If a load instruction loads the same data more than a given number of times (determined by a threshold), it is marked as possibly belonging to a spinning loop. If at some point in time, a marked load loads different data, then it is checked whether the new data was written by another core (using cache coherence information). If so, it was a spinning loop. Again, by keeping a timestamp at the first occurrence of a load, the total spinning time can be measured. Tian et al. implemented this technique in software, but it could also be done in hardware. Because the Tian et al. mechanism is simpler to implement (only keeping track of loads, no processor state monitoring), we consider this method for quantifying spinning overhead.

This discussion assumed lock-based critical sections. In the case of transactional memory, one could measure the execution time of a transaction, and when it is rolled back because of a conflict, the time spent in the transaction is added as a synchronization penalty.

### Yielding

Spinning consumes resources despite of the fact that the thread does not make forward progress. Synchronization libraries are therefore optimized to avoid spinning when a long waiting time is to be expected. Instead of spinning, the threads trying to acquire the lock or barrier are scheduled out, and are awoken when the synchronization condition is met. Like that, the operating system can schedule other threads on the core, or shut down a core if there are no available threads. Since this is also a form of synchronization penalty, we also measure the time a thread is scheduled out. This can be done in a straightforward way in the operating system. In this work, we refer to this effect as yielding.

### Cache Coherence

Cache coherence affects multi-threaded performance by invalidating local cache lines in private caches, which in its turn may induce additional L1 cache misses. However, a balanced out-of-order processor core can hide (most) L1 data cache misses very well [25], hence we do not account for cache coherence misses for speedup stacks. This may introduce error in case a workload suffers from a large number of L1 data cache misses along with long chains of dependent instructions which would prevent the out-of-order core from hiding their performance impact. However, in case L1 misses do incur a penalty (e.g., in an in-order architecture), coherence misses can be detected by noticing that invalidation by the coherence mechanism causes a cache line to be invalid without being (immediately) replaced by another cache line. Also, in case of an invalidation, usually only the status bits are adapted, while the tag remains in the tag array. If a miss occurs, but there is a hit in the tag array and the status is invalid, we can assume that this is most likely a coherence miss.

### Load Imbalance

The load imbalance component for a thread is computed as follows. Knowing the execution time of the slowest thread, we add a load imbalance component to each of the other threads such that the sum of all the cycle components for each thread equals the execution time of the slowest thread. This accounts for the load imbalance at the end of the (parallel part of the) program.

Imbalance at barriers is accounted as synchronization, either through spinning or yielding, as described earlier in this chapter. The reason is that it is impossible for the cycle accounting architecture when implemented in hardware to distinguish lock spinning from barrier spinning (or yielding). This problem can be solved though by computing speedup stacks for each region between consecutive barriers; the imbalance before each barrier (to be computed alike the load imbalance at the end of the program as described above) then quantifies barrier overhead.

### Hardware Cost of Profiling Tool

The total hardware cost of the counter architecture for measuring the impact of resource sharing is limited to 952 bytes per core:

- ATD: 32 (sampled sets) $\times$ 8 (associativity) $\times$ 27 bit (tag + replacement) = 864 byte;

- Marking (inter-thread) misses: 1 bit per ROB entry = 128 bit = 16 byte;
- The intra-thread latency counters in the MSHRs: 32 $\times$ 10 bit = 40 byte;
- ORA: 20 bit $\times$ 8 (banks) = 20 byte;
- The counters for the total number of accesses, the number of sampled accesses and the number of sampled inter-thread misses: 3 $\times$ 20 bit = 8 byte;
- The total interference cycle counter: 32 bit = 4 byte.

This hardware cost scales linearly with the number of cores. Even for a multi-core processor with a large number of cores, the total hardware cost is limited compared to the total transistor budget.

Not only is the amount of storage needed small, implementing the counter architecture should also be feasible in practice. The circuitry is localized to specific regions of the processor core. In particular, there is a set of counters in the ROB and MSHRs; there is the ATDs; and there is the ORAs. The counter architecture is unlikely to affect cycle time. The counters are read out by system software at regular (but coarse-grain) intervals, e.g., at the end of each timeslice.

The accounting for spinning overhead using the Tian et al. [60] approach incurs a load table: assuming a spinning loop contains at most 8 loads, 8 entries are needed in the table, containing the load PC, the address, the loaded data, a mark bit and a timestamp, which amounts to 217 bytes per core (assuming 64 bit addresses and data). The total hardware cost is therefore 1.1KB per core, or 18KB in total for a 16-core CMP.

The cycle component accounting architecture implemented in hardware provides raw cycle counts that are then processed in software. For example, for computing positive interference, the cycle accounting hardware architecture computes the total number of cycles a core is stalled on an LLC load miss plus the number of LLC load misses; system software then computes the average penalty per miss from these raw event counts and performs the interpolation as previously explained. This way, hardware complexity is limited and the proposed accounting architecture is feasible to implement in hardware.

| *Suite* | *Benchmark* | *Input* |
|---|---|---|
| SPLASH-2 | cholesky | tk29.O |
| | fft | 4M points |
| | lu.cont | 1024×1024 matrix |
| | lu.ncont | 1024×1024 matrix |
| | radix | 1M integers |
| | water-nsquared | 2197 molecules |
| | water-spatial | 2197 molecules |
| PARSEC | blackscholes | simsmall,simmedium |
| | bodytrack | simsmall |
| | canneal | simsmall,simmedium |
| | dedup | simsmall,simmedium |
| | facesim | simsmall,simmedium |
| | fluidanimate | simmedium |
| | ferret | simsmall,simmedium |
| | freqmine | simsmall,simmedium |
| | swaptions | simsmall,simmedium |
| Rodinia | bfs | 1M nodes |
| | heartwall | test.avi, 5 frames |
| | lud | 512×512 matrix |
| | needle | 4096×4096 matrix |
| | srad | 2048×2048 matrix |

**Table 6.1:** Considered benchmarks for speedup stacks.

### 6.2.2 Experimental Setup

This section describes the experimental setup used for generating the speedup stacks in the subsequent sections. We implemented the cycle accounting architecture in the gem5 simulator [6]. Further, we simulate a set of multi-threaded benchmarks from the SPLASH-2 [63], PARSEC [5] and Rodinia [13] benchmark suites that we were able to run in our simulation environment (see Table 6.1). All benchmarks were compiled using GCC version 4.3.2 with the `-O3` optimization level. Results are gathered from the parallel fraction of the benchmarks only. We simulated all benchmarks for 1, 2, 4, 8 and 16 threads, and we assume as many threads as there are cores, unless mentioned otherwise.

The simulated processor is a CMP consisting of four-wide superscalar out-of-order cores. L1 caches are private (32KB L1 I-cache and 64KB L1 D-cache), and the shared L2 cache is 2MB in size and is the

| | |
|---|---|
| Core frequency | 2GHz |
| Core pipeline width | fetch: 8; dispatch, issue and commit: 4 |
| ROB size | 128 entries |
| Load and store buffer | 96 entries each |
| Branch predictor | 64K entry tournament, 4K entry BTB |
| No. of cores | 1, 2, 4, 8, 16 |
| L1 I-cache | 32KB, 4-way, 64B line, 1 cycle |
| L1 D-cache | 64KB, 4-way, 64B line, 2 cycles |
| Shared L2 cache | 2MB, 8-way, 64B line, 10 cycles, 32 MSHRs |
| On-chip bus (L1 $\leftrightarrow$ L2) | 2GHz, 32 byte |
| Memory controller | FCFS, 16-entry write buffer |
| Memory bus | 1333MHz, 64 bit |
| DRAM | 667MHz DDR, 8 banks, 4KB row buffer |
| DRAM timing | 9-9-9-7 (tRP-tRCD-CL-CWL) |

**Table 6.2:** Simulated multi-core processor configurations for speedup stacks.

last-level cache (LLC) — the technique proposed in this work can be trivially extended to architectures with private L2 caches and a shared L3 LLC. All cores share the memory bus and the memory subsystem with 8 memory banks. Further details about the simulated processor configuration are listed in Table 6.2.

### 6.2.3 Validation

Validating an implementation for computing speedup stacks is challenging, because it is hard to isolate each of the stack contributors. We first validate how well our counter architecture estimates interference between threads and subsequently we evaluate the accuracy of the estimated speedup by a speedup stack versus the actual measured speedup for an application.

#### Validating the Counter Architecture

To evaluate the accuracy of the counter architecture we consider 19 SPEC CPU2006 benchmarks that properly run in our simulation environment. We use SimPoint [56] to select 1B-instruction representative samples from which we create a large number of multi-program workloads. The reason for choosing for a multi-program instead of a multi-threaded workload for the validation of our counter architecture is that in a multi-program workload threads only affect each other's performance due to resource sharing, and not because of synchronization. Furthermore, positive interference can neutralize negative interference in a multi-threaded application which makes evaluating the accuracy of the counter architecture more complicated.

For the evaluation of the accuracy of the counter architecture we define error as the relative difference between the estimated and the actual isolated execution times:

$$\begin{aligned} Error &= \frac{T_{isolated,estimated} - T_{isolated,\ measured}}{T_{isolated,\ measured}} && (6.5) \\ &= \frac{\left(T_{multi\text{-}core} - T_{estimated\ interference}\right) - T_{isolated,\ measured}}{T_{isolated,\ measured}}. && (6.6) \end{aligned}$$

A positive error implies an overestimation of the predicted isolated execution time or an underestimation of inter-thread interference; a negative error implies an underestimation of the predicted isolated execution time. We adopt the following simulation approach for computing the error. We first simulate a multi-program workload and stop the simulation when one of the programs has executed 1B instructions. We estimate the isolated execution times for each of the programs using the proposed counter architecture, i.e., $T_{isolated,estimated}$. We also determine the number of instructions executed for each program in the workload mix — one program has executed 1B instructions, the other programs have executed less than 1B instructions. For each program in the workload mix, we then run a single-threaded simulation for as many instructions as during multi-core execution, and we determine the isolated execution time, i.e., $T_{isolated,measured}$. This procedure guarantees that the same amount of work is done during multi-core execution as during isolated single-threaded execution for each program in the workload mix.

We evaluate the accuracy of the counter architecture in two steps. We first consider an idealized memory system and evaluate the counter architecture's accuracy for estimating the impact of inter-thread misses (additional misses in the shared cache) on overall performance. Secondly, we evaluate the counter architecture's overall accuracy while considering both inter-thread misses as well as waiting cycles on intra-

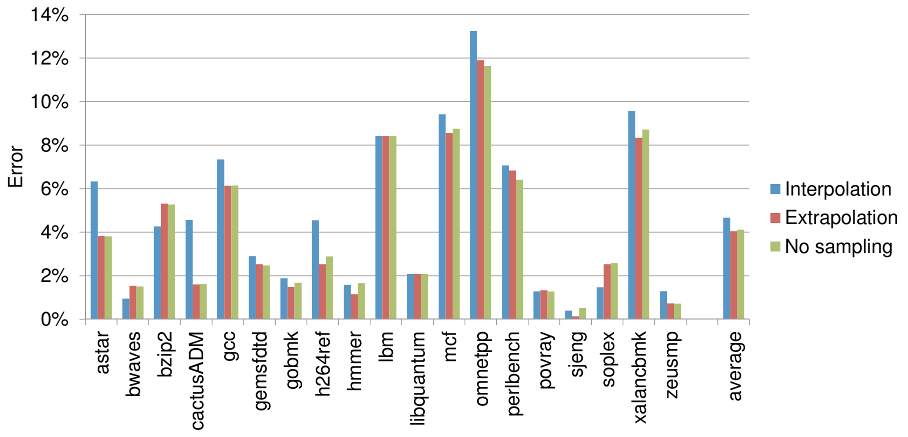


**Figure 6.3:** Average isolated execution time estimation error per benchmark for Equation 6.1 and 6.2 (eight cores, fixed memory latency, 32 sampled sets), compared to no sampling.

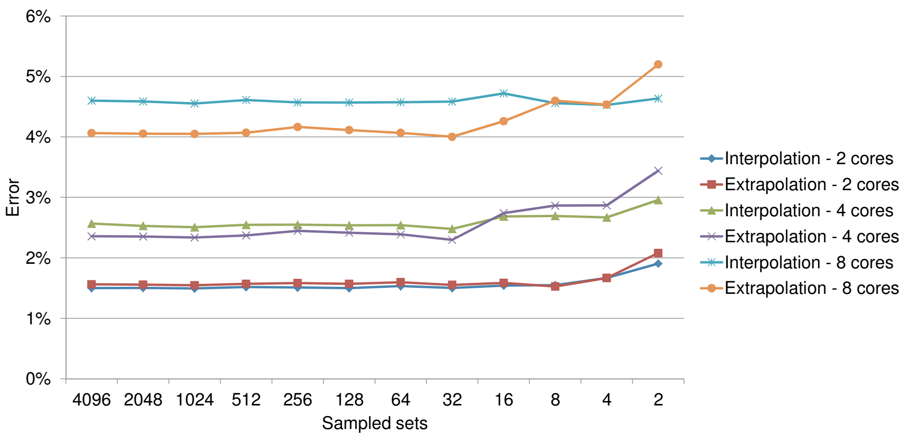


**Figure 6.4:** Average isolated execution time estimation error for the interpolation and extrapolation approaches as a function of the number of sampled sets; we assume fixed memory access latency.

thread misses; we consider a realistic memory system including hardware prefetching, memory banks and an open-page policy.

**Estimating the impact of inter-thread misses.** We first evaluate the counter architecture's accuracy for estimating the effect of inter-thread misses on overall performance. To this end, we consider an idealized memory system in order to focus on inter-thread misses and eliminate the effect of waiting cycles on intra-thread misses. We assume a fixed memory access latency (100ns) and assume there are no bank conflicts. Figure 6.3 shows the isolated execution time error when using the extrapolation and interpolation approaches (Equations 6.1 and 6.2, respectively). The extrapolation approach slightly outperforms the interpolation approach with an average error of 4.05% versus 4.66%, respectively.

Figure 6.4 shows the impact of sampling frequency in the ATD on accuracy for both approximations. The error of the interpolation approximation seems to be less sensitive to sampling frequency compared to the extrapolation approach. The reason is that the interpolation approach measures the penalty of all misses to infer the penalty for the inter-thread misses, whereas the extrapolation approach measures the penalty of the sampled inter-thread misses only and then extrapolates to all inter-thread misses. At a low sampling frequency, the penalty for the sampled inter-thread misses is not representative for the other inter-thread misses, hence accuracy degrades. We find that 32 sampled sets is a good design point for two, four and eight cores.

**Overall accuracy evaluation.** In the previous section, we assumed idealized memory (fixed access latency) in order to evaluate the counter architecture's accuracy with respect to estimating the impact of inter-thread misses. We now consider a realistic memory system with multiple banks along with an open-page policy; further, we assume hardware prefetching is enabled. This allows for evaluating the accounting architecture's ability to accurately estimate the effect of both inter-thread misses and waiting cycles. Figure 6.5 shows the measured versus estimated interference for two, four and eight cores. The counter architecture achieves an average (absolute) error of 3.75% for 2 cores, 5.57% for 4 cores, and 14.2% for 8 cores. This is fairly accurate given the level of interference, which equals 9.3% for 2 cores, 19.5% for 4 cores, and 55.4% for 8 cores — in other words, the accounting architecture captures 74.3% of the inter-thread interference on average for 8 cores. Note that although the absolute error increases with increasing core count, e.g., it increases from 5.57% for 4 cores to 14.2% for 8 cores, the relative error compared to the level of interference actually decreases from 28.5% for 4 cores to 25.6% for 8 cores. In other words, absolute error increases with core count but so does the level of interference, hence in the end, the relative accuracy of the counter architecture is fairly sta-

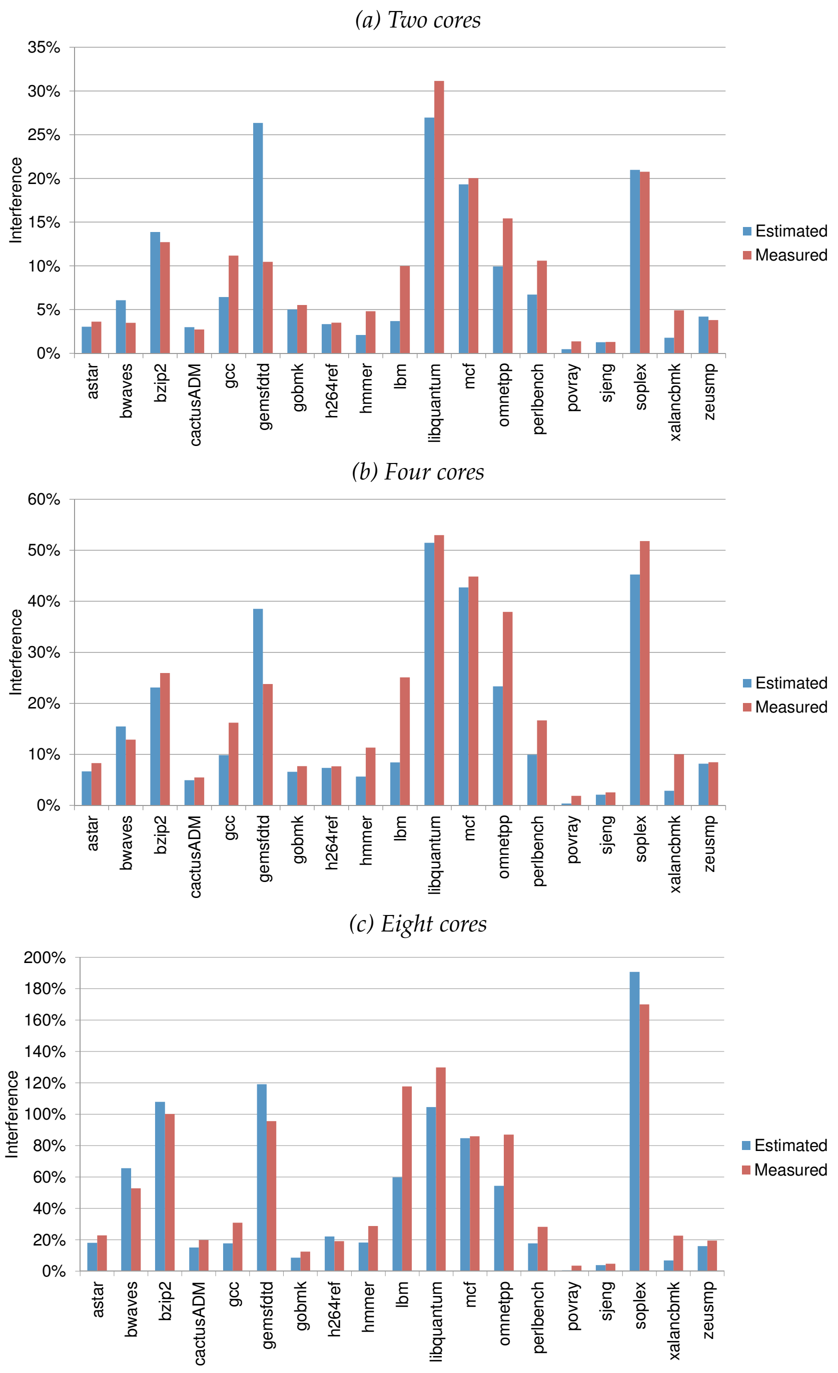


**Figure 6.5:** Estimated versus measured interference for (a) a dual-core, (b) a quad-core, and (c) an eight-core system.

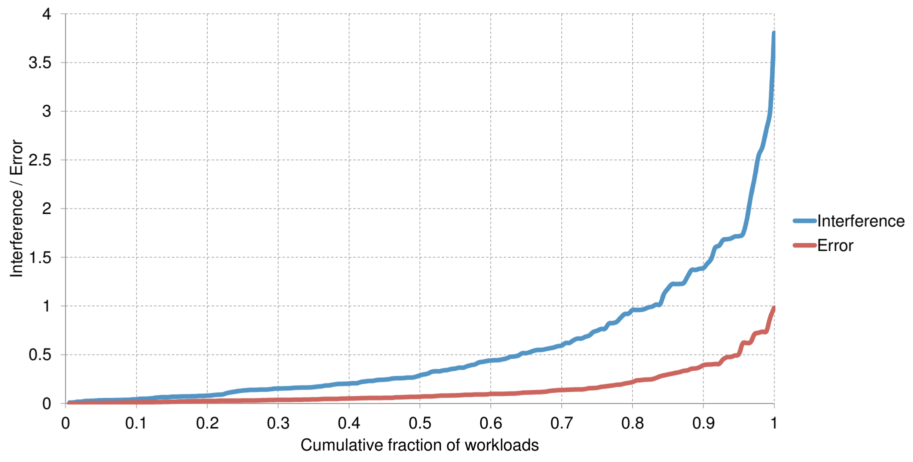


**Figure 6.6:** Interference and error for estimating isolated execution time for an 8-core processor; workloads are sorted along the horizontal axis.

ble and actually decreases with core count, i.e., the proposed counter architecture is able to consistently capture the most significant sources of inter-thread interference. Further, we expect absolute accuracy to improve given microarchitecture enhancements that reduce inter-thread interference, such as multiple memory controllers, limiting the number of cores that share a cache, etc.

The results shown so far presented error numbers that are averaged across a number of multi-program workloads, e.g., there are 19 2-program workloads and 10 4-program and 10 8-program workloads per benchmark. Figure 6.6 shows the same data but does not average out across a number of multi-program workloads; i.e., there is a data point for each workload (190 in total). The graph shows a cumulative distribution for the interference and the error: the horizontal axis shows the fraction of workloads for which the interference and error are below the corresponding value on the vertical axis. This graph shows the amount of variation in both the interference and the error. We observe that interference can be very high for some workloads, up to 3.8×. The proposed cycle accounting architecture accurately identifies the workloads with high interference levels, and the error is substantially lower compared to the level of interference. In particular, for 80% of the workloads, interference is as large as 92%, yet the error is less than 21%; similarly, for 70% of workloads, interference is as large as 59% with an error below 13.6%. This graph shows once more that the accounting archi-

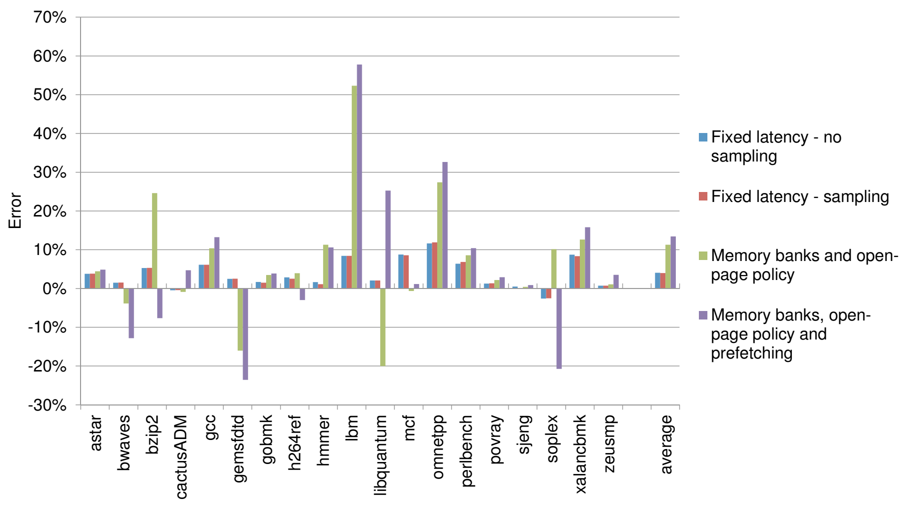


**Figure 6.7:** Error analysis per benchmark for an 8-core processor.

tecture is able to measure a large fraction of the interference in all cases: a higher interference results in a higher absolute error, but the relative error remains approximately the same (around 25%).

**Error analysis.** In order to better understand the different sources of error, we set up a number of experiments and we quantified how the error is affected by various sources of interference for the various benchmarks, see Figure 6.7. For the first bar, we assumed no bank nor row buffer conflicts (fixed memory access latency), no hardware prefetching and full (i.e., non-sampled) ATDs. In this case, the accounting architecture achieves an average absolute error of 4.1% on an 8-core system. This error follows mainly from second-order effects that are not captured. For example, intra-thread waiting cycles hidden underneath inter-thread misses during multi-core execution are possibly not hidden in isolated execution; in this case, the accounting architecture would account for the waiting cycles, although it should not. We chose not to consider second-order effects in order not to complicate the accounting architecture hardware.

The second bar shows the impact of sampling only 32 sets of the cache. This has no noticeable effect on accuracy. Adding banks and considering an open-page policy (third bar) increases the average error to 11.5%, and adding prefetchers (fourth bar) increases the overall error to 14.2%. These errors also stem from second-order effects that

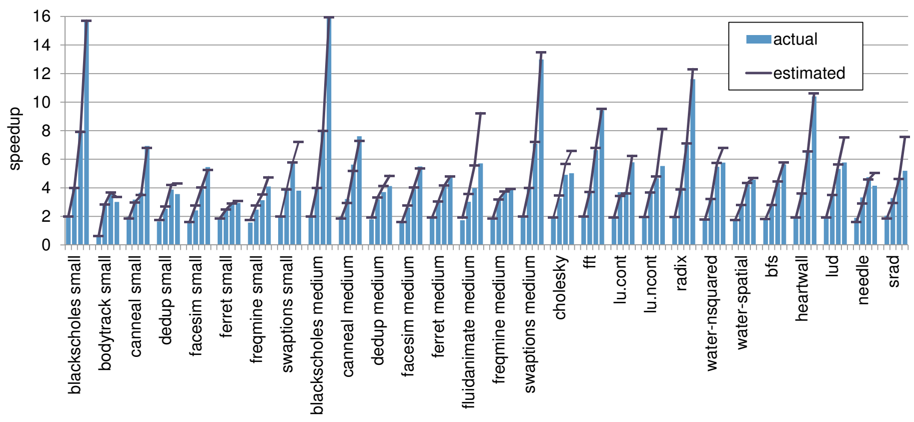


**Figure 6.8:** Actual speedup and estimated speedup for all benchmarks for 2, 4, 8 and 16 threads.

are not modeled in our cycle-accounting method, to reduce the overhead and complexity of the hardware additions. For example, when an inter-thread miss causes waiting cycles (e.g., due to a bank conflict) for an intra-thread miss of the same thread, these cycles are not accounted since both misses belong to the same thread. Accounting for this would require a categorization of every miss into inter- or intra-thread, which is impossible using a sampled ATD, and would also need communicating the inter- and intra-thread miss information to the memory controller, which complicates the design.

Furthermore, Figure 6.7 also shows that the resulting error can be positive or negative (we took the absolute values to calculate the averages, such that these errors do not compensate for each other). This shows that our technique is not biased, i.e., there is no consistent under- or overestimation. As a result, additional accounting to decrease the error in the case interference is overestimated tends to also increase the error in case the interference is underestimated and vice-versa. This means that in order to reduce the error consistently, both underestimation and overestimation cases needs to be handled, which would make the accounting architecture overly complex.

## Validating Estimated Speedup

After validating our counter architecture, we now look at the accuracy of the estimated speedup $\hat{S}$ from a speedup stack versus the actual speedup $S$ for an application. We define error as

$$Error = \frac{\hat{S} - S}{N}, \tag{6.7}$$

with $N$ the number of cores/threads.

Figure 6.8 shows the actual speedup against the estimated speedup for 2, 4, 8 and 16 threads/cores. Accuracy is fairly good: cycle component accounting identifies the benchmarks that scale well versus the benchmarks that do not, and more importantly, the method fairly accurately identifies the degree of scaling. The average absolute error is 3.0%, 3.4%, 2.8% and 5.1%, for 2, 4, 8 and 16 threads, respectively.

For some benchmarks through, the estimated speedup is off, see for example fluidanimate medium (22.0%), swaptions small (21.3%), lu.ncont (16.2%) and srad (14.8%). The reason for these errors are multifold. For one, as mentioned before, the proposed method does not account for parallelization overhead. Computing the increase in dynamic instruction count for a multi-threaded execution over single-threaded execution, and subtracting the number of instructions due to spinning, is a measure for the amount of parallelization overhead. We found parallelization overhead to be fairly high for swaptions small (26% more instructions) and fluidanimate medium (18% more instructions). Other possible reasons for the error are inaccuracies for estimating the impact of positive and negative interference (cf. interpolation and extrapolation); inaccuracies for computing spinning overhead as it is based on some threshold; not taking into account the impact of cache coherence.

### 6.2.4 Applications

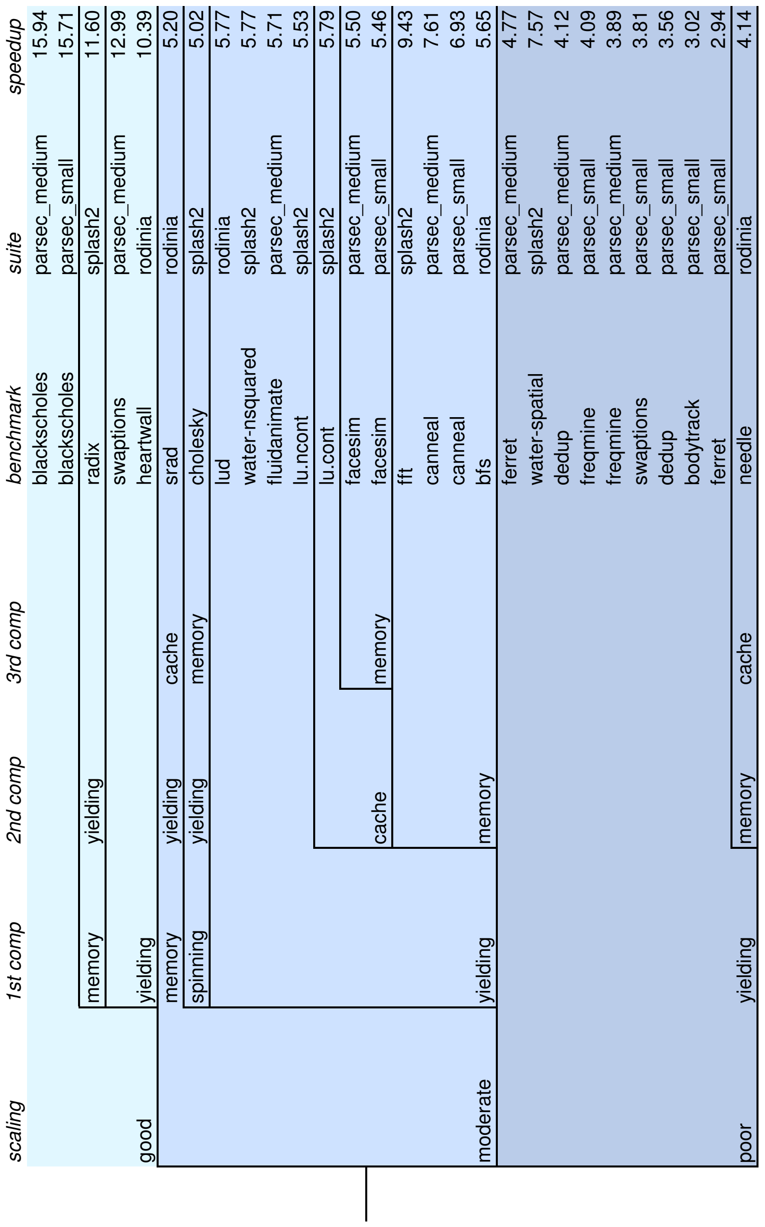
Having obtained confidence in the accuracy of speedup stacks, we now explore a number of applications to illustrate their usefulness for performing performance analysis and workload characterization studies, as well as for driving hardware and software optimizations.

#### Benchmark Classification

After generating speedup stacks for a large number of benchmarks, one can classify the benchmarks according to their scaling behavior. Figure 6.9 shows such a schematic representation for all the benchmarks considered in this study, based on the speedup stacks for 16-threaded execution. This tree-like representation is built up as follows. Going from left to right, we first classify the benchmarks according to their scaling behavior. Good scaling behavior means a speedup of at least 10× for 16 threads, while poor scaling benchmarks have a speedup of less than 5×. The in-between benchmarks are classified as moderate. The next bifurcation in the tree is based on the largest scaling delimiter (i.e., the largest component in the speedup stack). It shows the main scaling delimiters for each of the benchmarks that belong to a certain scaling category. The type of the component always appears on top of the line. If there is no component on top of the line, then there is no considerable component that limits scaling (e.g., for blackscholes, as discussed before). The following two bifurcations reflect the second and third largest scaling bottlenecks. Again, if there is no component name on top of the line, then all remaining components are negligible. The fifth column lists the names of the benchmarks, followed by the benchmark suite they belong to and their actual speedup number. Figure 6.10 shows speedup stacks for a subset of the benchmarks. For some applications the base speedup differs slightly from the reported speedup in Figure 6.9 because the speedup stacks show estimated speedup while the other figure shows actual measured speedup.

An insightful way for reading the tree graph in Figure 6.9, is to read the graph from right to left. To find the characteristics of a specific benchmark, locate the benchmark at the right handside of the figure. Then follow the first line that is underneath this benchmark to the left to find out the main scaling limiting components and its scaling category. For example for facesim, the major scaling bottlenecks are, in decreasing order, yielding, LLC interference and memory interference, while achieving moderate scaling.

There are a number of interesting observations to be made from this tree-based classification. First, only few benchmarks scale well: 5 out of the 28 benchmarks have a speedup of at least 10× for 16 threads. The other two categories (moderate and poor scaling) contain approximately the same number of benchmarks, but by looking at the speedup numbers, it is clear that even in the moderate group, most of the benchmarks achieve a speedup only slightly above 5×. The poorest performing benchmark (ferret) shows a speedup of less than 3× for 16 threads. It is also interesting to note that scaling behavior typically improves with input size, see swaptions as an extreme example (speedup increases from 3.8× to 13.0× when the simmedium input is used compared to simsmall); this illustrates the weak scaling behavior of this



| *scaling* | *1st comp* | *2nd comp* | *3rd comp* | *benchmark* | *suite* | *speedup* |
|---|---|---|---|---|---|---|
| | | | | blackscholes | parsec_medium | 15.94 |
| | | | | blackscholes | parsec_small | 15.71 |
| | memory | yielding | | radix | splash2 | 11.60 |
| | | | | swaptions | parsec_medium | 12.99 |
| good | yielding | | | heartwall | rodinia | 10.39 |
| | memory | yielding | cache | srad | rodinia | 5.20 |
| | spinning | yielding | memory | cholesky | splash2 | 5.02 |
| | | | | lud | rodinia | 5.77 |
| | | | | water-nsquared | splash2 | 5.77 |
| | | | | fluidanimate | parsec_medium | 5.71 |
| | | | | lu.ncont | splash2 | 5.53 |
| | | | | lu.cont | splash2 | 5.79 |
| | | | | facesim | parsec_medium | 5.50 |
| | | cache | memory | facesim | parsec_small | 5.46 |
| | | | | fft | splash2 | 9.43 |
| | | | | canneal | parsec_medium | 7.61 |
| | | | | canneal | parsec_small | 6.93 |
| moderate | yielding | memory | | bfs | rodinia | 5.65 |
| | | | | ferret | parsec_medium | 4.77 |
| | | | | water-spatial | splash2 | 7.57 |
| | | | | dedup | parsec_medium | 4.12 |
| | | | | freqmine | parsec_small | 4.09 |
| | | | | freqmine | parsec_medium | 3.89 |
| | | | | swaptions | parsec_small | 3.81 |
| | | | | dedup | parsec_small | 3.56 |
| | | | | bodytrack | parsec_small | 3.02 |
| | | | | ferret | parsec_small | 2.94 |
| poor | yielding | memory | cache | needle | rodinia | 4.14 |

**Figure 6.9:** Tree graph showing main speedup delimiter components for each benchmark for 16 threads: follow the first line underneath a benchmark from right to left to find its scaling behavior and its third, second and first (from right to left) largest components.

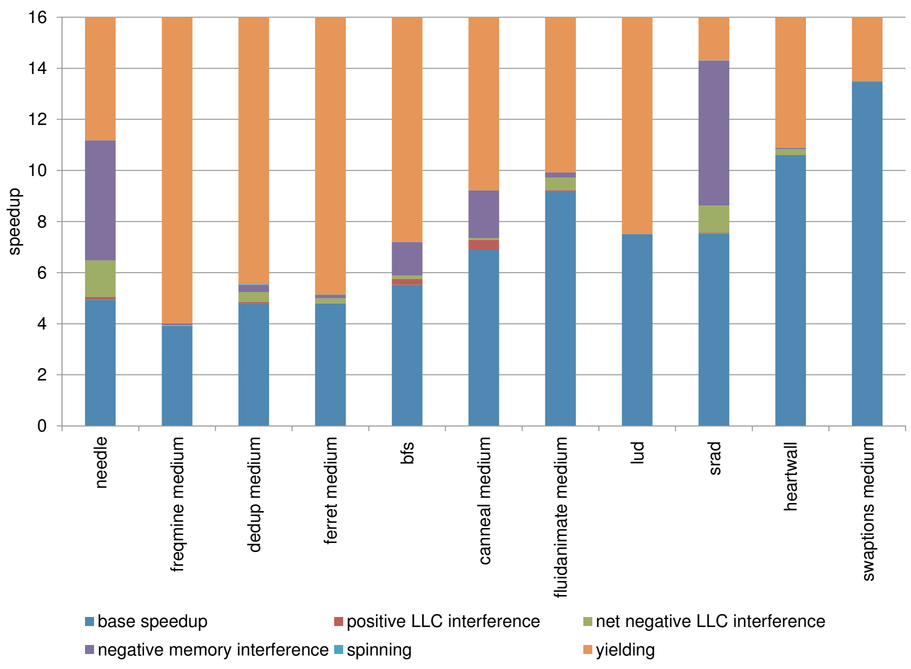


**Figure 6.10:** Speedup stacks for a selection of benchmarks with 16 threads.

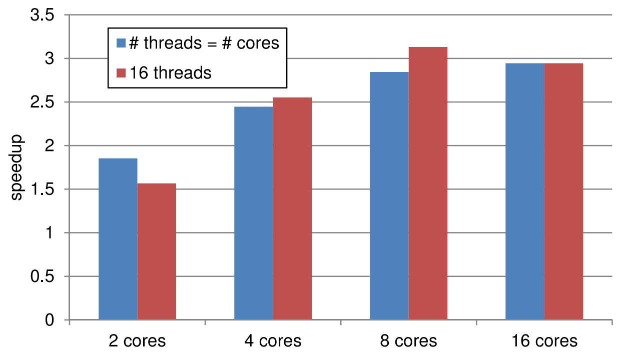


**Figure 6.11:** Speedup numbers for ferret as a function of the number of cores. The number of threads equals the number of cores (left bars) or equals 16 (right bars).

workload.

Interestingly, yielding seems to be the most significant scaling delimiter (see also the speedup stacks in Figure 6.10). It is the largest component for 23 of the 28 benchmarks (see the second column from

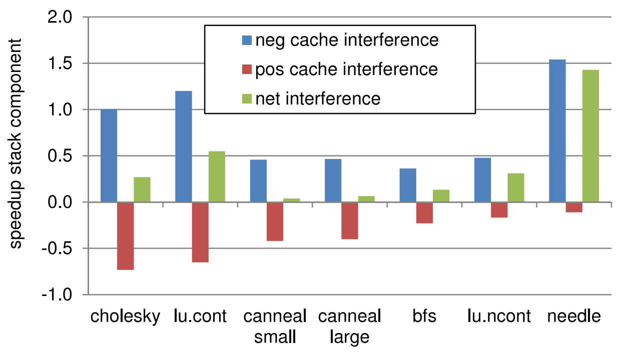


**Figure 6.12:** Negative, positive and net LLC interference components.

the left), and the second largest component for 3 of the remaining 5 benchmarks. For 13 benchmarks it is the only component with a non-negligible value, which means that the only limiting factor is the fact that only a few threads are active at a time. In this case, the speedup number is an approximation of the average number of active threads. This means that, although there are 16 threads spawned, only a fraction of the threads is active, and hence only a fraction of the cores are busy. This suggests that these benchmarks do not need 16 cores; hence, a number of cores that is slightly larger than their speedup number might yield the same performance. This insight is validated in Figure 6.11, which compares speedup for the 16-threaded version of ferret run on 2, 4, 8 and 16 cores. It reveals that if there are 16 threads spawned, performance saturates at 8 cores (the lower performance for 16 cores can be explained by the Linux scheduler being less efficient when there are more cores). The graph also shows the speedup when the number of threads equals the number of cores, from which it follows that having more software threads than hardware thread contexts (cores) leads to better performance.

### Understanding LLC Performance

Sharing the LLC has two main advantages: cache space is used more efficiently compared to private caches, and shared data has to be fetched only once, and then it can be used by all cores (positive interference). This comes at the cost of negative interference (threads evicting each other's data). In this section, we investigate the impact of positive versus negative interference.

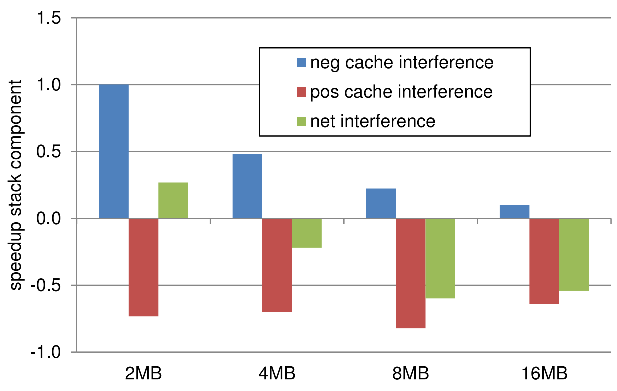


**Figure 6.13:** Negative, positive and net interference components for cholesky as a function of LLC size.

Figure 6.12 shows the negative, positive and net interference components in the LLC assuming 16 cores; only the benchmarks with a non-negligible positive interference component are shown. For all benchmarks, the negative interference exceeds the positive interference, resulting in a net component that has a negative impact on performance (which is the green component in Figure 6.10).

However, as we enlarge the LLC, negative interference should decrease (fewer capacity misses), while positive interference should remain constant (since this is a result of the characteristics of the program, not the hardware). This is validated in Figure 6.13, where we show the same components for cholesky for an LLC of 2MB (default), 4MB, 8MB and 16MB. Negative interference indeed decreases, while positive interference remains approximately constant as a function of cache size, resulting in a smaller net interference component, and even a negative one, which means that the total impact of cache sharing is positive for performance.

## 6.3 Speedup Stacks Measured in Software

We now discuss our second version of speedup stacks that is targeted towards managed language applications. In this version of speedup stacks we include a set of scaling delimiters that is tailored towards managed language applications, which is garbage collection, sequential parts of the application, thread imbalance, synchronization and hardware interference.

### 6.3.1 Constructing Speedup Stacks

For constructing those speedup stacks and measuring the impact of this set of scaling components on performance, we reuse our software profiling tool that we designed for generating bottle graphs (see Section 5.2). This allows us to generate speedup stacks for unmodified applications running on native hardware, incurring a low overhead. For generating speedup stacks we had to extend this profiling tool. How we modified this tool, so that it is also able to measure the various scaling delimiters for managed runtime applications, will be discussed in the following sections.

#### Garbage Collection

We construct speedup stacks for Java applications running with a stop-the-world garbage collector (meaning either the garbage collector or the application is running, but never both at the same time) and with a concurrent garbage collector (meaning parts of the garbage collection can be done while the application is running). We first discuss how we measure the impact on the performance of an application when using a stop-the-world garbage collector.

For constructing a speedup stack, we need to know the total time that the garbage collection threads pause the application. For a stop-the-world collector this value can be read out of a bottle graph, because it is the sum of the heights of all garbage collector boxes. This is due to the fact that garbage collection threads can only run concurrently with each other. Therefore, summing these heights leads to the total time garbage collection was running and thus halted the application threads.

For a concurrent garbage collector we can not follow the same approach, because garbage collector threads can now run concurrently with the application threads. However, at some points during the execution of an application a concurrent collector can also halt the application threads and thus behaves like a stop-the-world garbage collector, for example, when dealing with a full heap and the application can not allocate objects any more. We modify the JVM so that it signals when concurrent GC becomes stop-the-world garbage collection. In our profiling tool we then record the timing of these periods.

### Synchronization

When threads have to wait due to synchronization they use a futex system call. We already intercept this system call with our profiling tool because it causes a thread to change its state and therefore delineates a time interval over which we calculate our criticality metric. We extend the profiling tool to measure the time threads are waiting inside this system call. This value is needed by our speedup stacks to quantify the impact of synchronization on application speedup.

### Sequential Parts of the Application

In Jikes RVM, there is a service thread called the MainThread that performs the sequential part of the application, namely initializing the JVM and data, performing initial compilation, spawning the application threads, and later performing shutdown activities. Therefore, the time this thread is active is considered as the sequential part of an application. This value can be read out of a bottle graph by looking at the height of the box of the MainThread.

### Thread Imbalance

Thread imbalance happens when application threads do not finish their execution at the same time. As soon as one thread finishes its execution, the thread starts waiting inside an exit system call until all threads have finished their execution. This is similar to what happens with barrier synchronization between threads. To account for this waiting time, we extend our profiling tool to measure the time threads spend waiting inside an exit system call.

### Hardware Interference

The impact of hardware interference on the performance of an application is very difficult to measure in system software. Though we cannot precisely measure this overhead, we present this component as the difference between the measured speedup and the ideal speedup, minus all the other overhead components. We also explain this remaining component by analyzing the application behavior through hardware performance counters. Because we have the advantage of running un-

| Benchmark | Suite | Version | Overhead |
|---|---|---|---|
| lusearch | DaCapo | 2009 | 1.15% |
| pmd | DaCapo | 2009 | 0.53% |
| sunflow | DaCapo | 2009 | 1.04% |
| xalan | DaCapo | 2009 | 0.40% |

**Table 6.3:** Considered benchmarks for speedup stacks measured in software.

modified managed applications on current hardware, we can quickly and accurately gather statistics on real program behavior.

### 6.3.2 Experimental Setup

Our experimental setup used for this version speedup stacks is similar to the experimental setup that we used for bottle graphs and discussed in Section 5.2. We use the same hardware platform but the evaluated set of benchmarks is different from before. We again use DaCapo benchmarks, but only a subset of them. Table 6.3 shows the benchmarks we evaluate in this study. The reason for choosing only these applications is that speedup stacks only make sense for multi-threaded Java applications, and the other two multi-threaded applications we evaluated with bottle graphs, namely avrora and pseudoJBB, do not allow to vary the number of application threads without changing the input size.

We again use Jikes Research Virtual Machine version 3.12 and do experiments with the default best-performing garbage collector on Jikes, the stop-the-world parallel generational Immix collector. We also evaluate applications running with a concurrent garbage collector. Jikes RVM uses a mark-sweep snapshot-at-the-beginning concurrent GC algorithm. The concurrent collector initiates a new collection cycle with a trigger defined per benchmark, i.e., a quantity of memory in bytes is specified, and after this amount of allocation, a new concurrent collection cycle is triggered. This concurrent collector requires a small pause of the application to first identify a consistent root set, and later to actually free memory. Between these two actions, the collector (all GC threads) runs concurrently with the application threads in order to trace the object graph and find reachable objects in order to identify live data. All objects that are not marked as live are then freed by the collector, while the application is paused. In addition, during the concurrent activity, all application writes go through a barrier to coordinate with the GC so that they are not writing to the same object, and so the GC

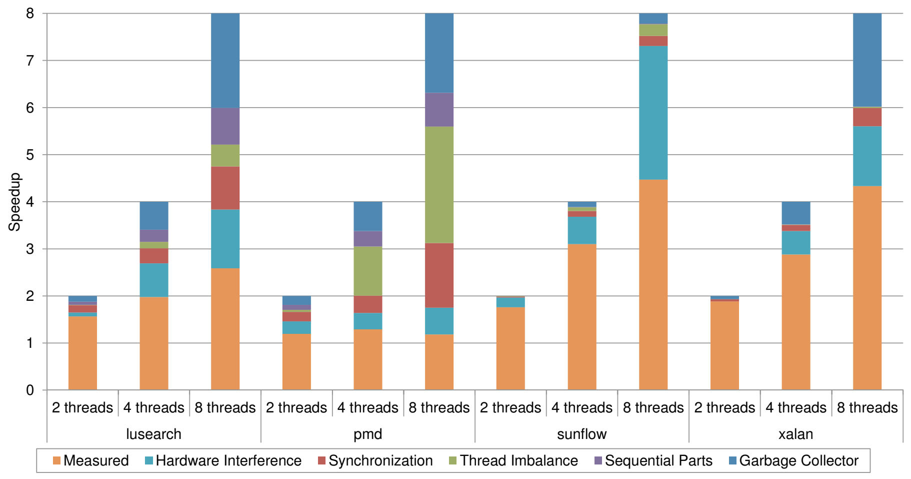


**Figure 6.14:** Speedup stacks for all applications with a stop-the-world garbage collector (2×minimum heap size).

maintains a consistent view of heap pointers [7].

### 6.3.3 Java Application Scaling Analysis

For understanding how the whole application scales and the relative contributions of various multi-threaded performance deficiencies, we now use the speedup stacks. In the next sections we will analyze the performance of managed language applications running on Jikes RVM with a stop-the-world and a concurrent garbage collector. We always use two GC threads, which was identified to be optimal for Jikes RVM in the bottle graphs study in Chapter 5.

#### Stop-the-world Garbage Collection

Figure 6.14 shows speedup stacks for all our multi-threaded benchmarks with 2, 4 and 8 application threads as compared to their single-threaded versions. The orange component at the bottom of each stack shows the measured speedup between single-threaded and multi-threaded execution and the colored boxes on top of it show the various speedup delimiters and their impact on speedup.

From the stacks we can see that most of our applications do not scale very well, although our bottle graphs showed that the individual

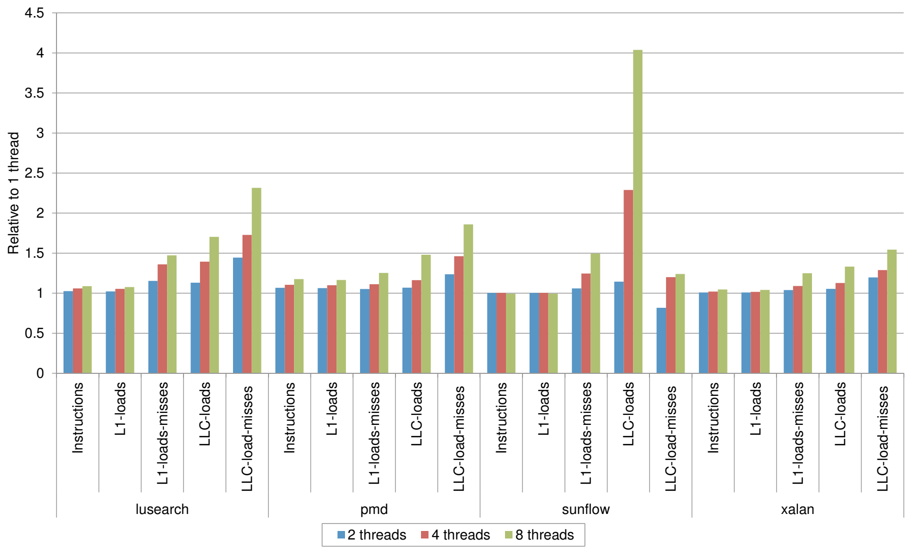


**Figure 6.15:** Data from hardware performance counters for all applications using a stop-the-world garbage collector. (Data normalized to one application thread.)

threads scale very well (see for example xalan in Figure 5.2). Applications sunflow and xalan show comparable speedup results, but the reason why their speedup is limited is different. While sunflow mostly suffers from interference in the underlying hardware, xalan mostly suffers from the limited garbage collector scalability. Pmd is the application that scales the worst, which we could already expect from our previous study of this benchmark with bottle graphs. The imbalance between the threads is the largest speedup delimiter, but the garbage collector and synchronization between the threads also have a large impact on speedup. The reason why lusearch does not scale well is a combination of all components in the speedup stack. Again the garbage collector is the main reason, together with interference between the threads in the hardware, or parallelism overhead.

To better understand the component for interference in the hardware, we look at data gathered from hardware performance counters in Figure 6.15. For sunflow, the application with the largest interference component, we see that the number of last level cache (LLC) loads goes up very fast when scaling to 4 and 8 application threads, while the number of LLC load misses does not increase significantly. For other benchmarks the interference in the hardware translates into a combination of

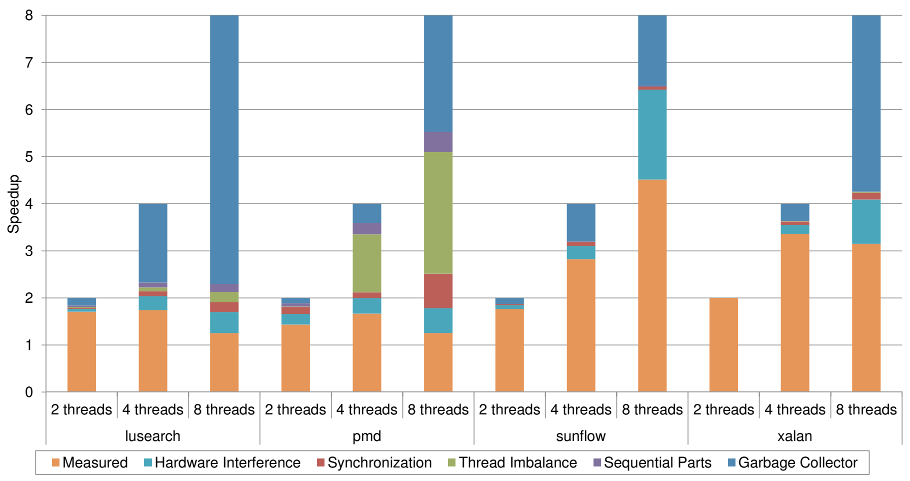


**Figure 6.16:** Speedup stacks for all applications with a concurrent garbage collector (2×minimum heap size).

increased number of LLC loads and LLC load misses, particularly for lusearch. The figure also reveals that the number of instructions stays almost constant when increasing the number of application threads, meaning that speedup is not limited because of additional instructions (called parallelization overhead) necessary for creating threads, doing synchronization, etc.

## Concurrent Garbage Collection

From concurrent garbage collection, it is expected that speedup is less limited by garbage collection as it is the case with stop-the-world garbage collection, because part of the garbage collection can now be done while the application threads are running and therefore the garbage collector should be less determinative to execution time than it is the case with a stop-the-world garbage collector. Figure 6.16 shows speedup stacks for our applications running with a concurrent garbage collector. In this experiment we use the same heap sizes for the applications as in the previous section which is two times the minimum heap size. As opposed to what we expected, the speedup stacks reveal that for all benchmarks the impact of the garbage collector has become larger compared to stop-the-world garbage collection.

The application that suffers the most from the garbage collector

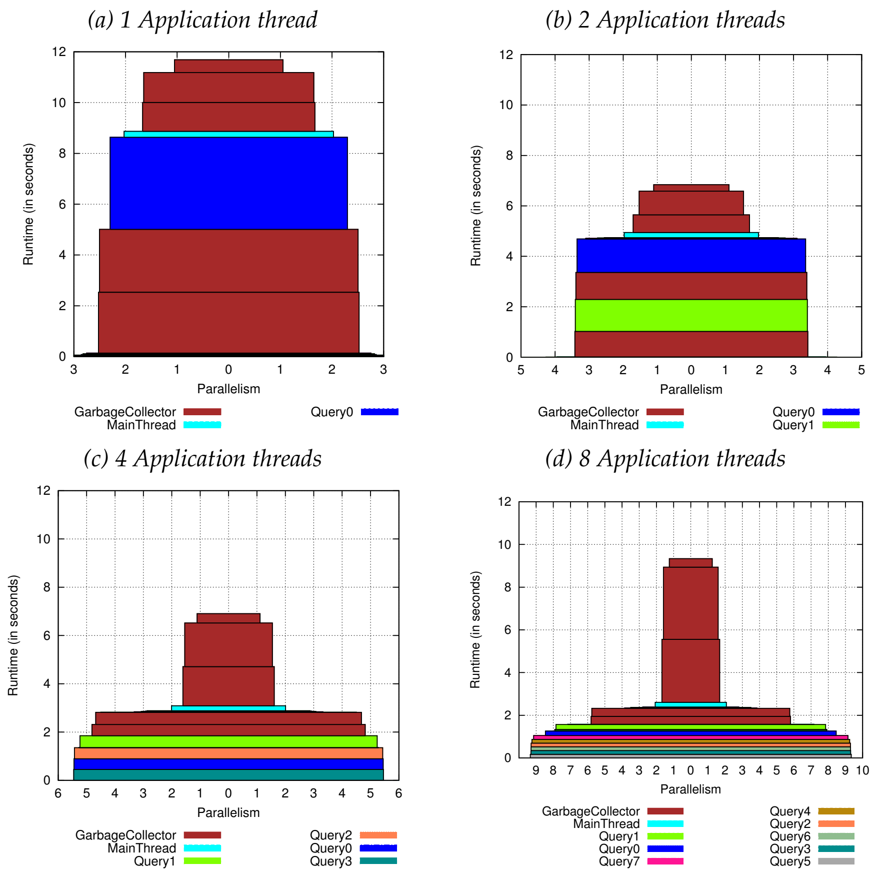


**Figure 6.17:** lusearch: scaling of application threads with concurrent garbage collector (2×minimum heap size).

is lusearch. This means that the concurrent garbage collector pauses the application threads very often, leading to a lot of stop-the-world phases for the application. This can also be seen from the bottle graphs in Figure 6.17. We note that in Jikes RVM, when we specify that there should be two garbage collection threads, the runtime environment spawns two threads to perform stop-the-world pauses, and two that perform concurrent tracing activities, in addition to having a main collector thread that performs some initialization work. Thus, in the bottle graphs of applications using the concurrent collector, there are five GC thread boxes. The graphs show that three GC threads always have limited parallelism (the stop-the-world ones and the initialization thread), while two have parallelism approaching that of the application

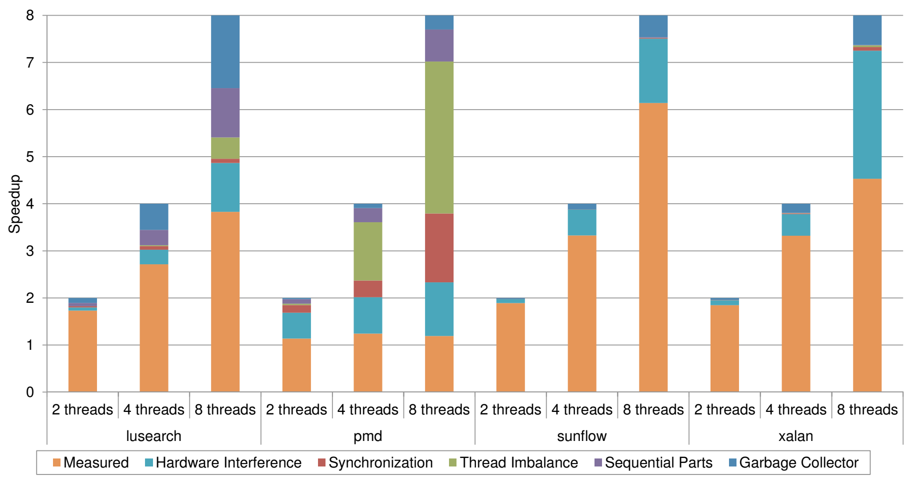


**Figure 6.18:** Speedup stacks for all applications with a concurrent garbage collector (10×minimum heap size).

threads (the concurrent GC threads). The height of the stop-the-world GC boxes, especially for 4 and 8 application threads, reveals that these threads have a high share of the execution time and are therefore very critical. From this we conclude that the concurrent garbage collector in Jikes RVM does not scale well, especially with small heap sizes.

To explore the scalability of the concurrent garbage collector when it is not constrained, we present speedup stacks in Figure 6.18 when the applications are run with a larger heap size (10×minimum heap size). For all applications, the impact of the GC's scalability on speedup is significantly reduced and the measured speedup improved, compared to Figure 6.16 (except for pmd that still suffers from large thread imbalance). For sunflow and xalan, the main speedup delimiter is interference in the hardware, as expected because of more threads concurrently running, while for lusearch it is a combination of different components.

For explaining the interference in hardware, we again present measurements from the hardware performance counters in Figure 6.19 when running with a concurrent collector and a large heap. Lusearch suffers from an increased number of LLC loads as the application thread count increases, and LLC load misses increase from 2 to 4 application threads, but go down for 8 threads. With a larger number of application threads, the positive interference between the threads in the LLC (meaning threads load data in the shared cache that can later

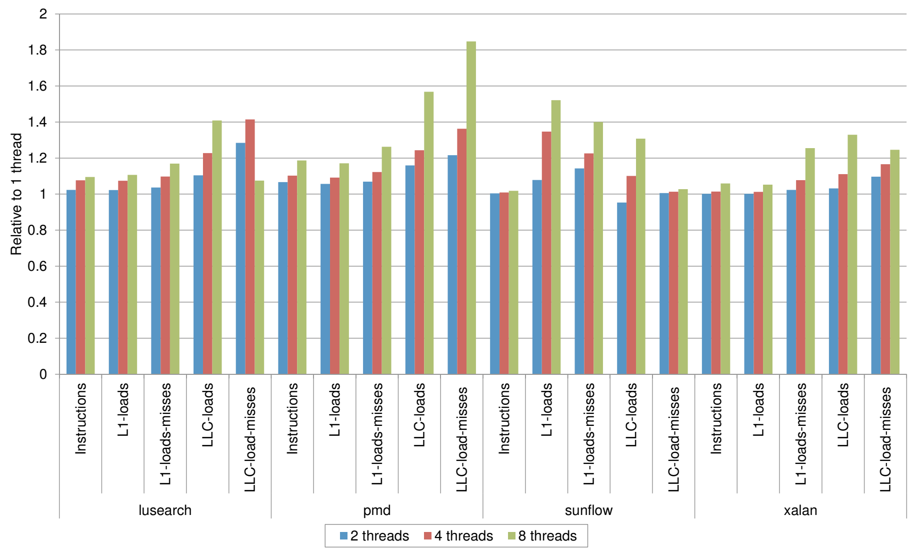


**Figure 6.19:** Data from hardware performance counters for all applications using a concurrent garbage collector with a large heap. (Data normalized to one application thread.)

be used by other threads) compensates almost completely for the negative LLC interference. Interestingly, this was not the case when using a stop-the-world collector (see Figure 6.15), which can disrupt the LLC and incur more LLC load misses for the application, especially at high thread counts and with a smaller heap size.

Sunflow has an increasing number of L1 loads, misses and LLC loads, which are the main causes of the hardware interference. This behavior is somewhat different than when we run sunflow with a stop-the-world garbage collector which does not have an increasing number of L1 loads as the application thread count increases. The increased number of L1 loads are due to the garbage collector accessing the L1 cache more often, but because the garbage collector is concurrent, the application does not suffer too much (and we see roughly equal numbers of L1 load misses as with a stop-the-world collector). When using the concurrent collector, we see that sunflow has fewer LLC loads than with the stop-the-world collector, but that is because we also use a larger heap size.

Xalan with a concurrent collector shows an increasing number of L1 load misses, which results in larger amounts of LLC load accesses and LLC load misses. This behavior is similar to using a stop-the-world garbage collector. Pmd's trends in Figure 6.19 also follow those with a stop-the-world GC and those of xalan; increased L1 load misses result in more LLC loads and load misses, even when using concurrent GC.

## 6.4 Summary

In this chapter, we showed that speedup stacks facilitate the visualization of performance and scalability bottlenecks in multi-threaded applications. We presented two versions of speedup stacks, one version that is measured in hardware and another version that is measured in software. We used the first version of speedup stacks for identifying scaling bottlenecks in a set of SPLASH-2, PARSEC and Rodinia benchmarks, for classifying the benchmarks based on their scaling delimiters, and for understanding LLC performance. The second version of speedup stacks, that is measured in software, targets managed language applications. They not only reinforce what we discovered with bottle graphs in the previous chapter (as in the case of pmd), but also reveal the impact of the limited scalability of the garbage collector, synchronization activities between application threads, imbalance of application threads, and the effect of hardware interference in the memory subsystem (which can be explained by performance counter data).

# Chapter 7

# Dynamic Performance Optimization

*In this chapter we show how our methods, besides useful for program analysis, can also be used to dynamically optimize applications' performance.*

## 7.1 Introduction

In the previous chapters, we used our methods for analyzing the performance and scalability of multi-threaded applications. However, we can also use them to dynamically optimize performance. In this chapter we optimize applications in three dimensions.

First, we propose a mechanism to dynamically optimize fairness by an improved scheduling of threads on a multi-core processor, which is not trivial, because interference between threads can lead to certain threads making faster progress than others.

Secondly, we design an algorithm to dynamically optimize the performance of multi-threaded applications, hereby reducing total execution time of the application. We also compare the performance of our algorithm to closely related work, called Bottleneck Identification and Scheduling (BIS) [34].

Thirdly, we show how energy consumption of a multi-threaded application can be reduced, which is becoming an important design concern for mobile and server market hardware.

## 7.2 Improving Multi-Core Scheduling

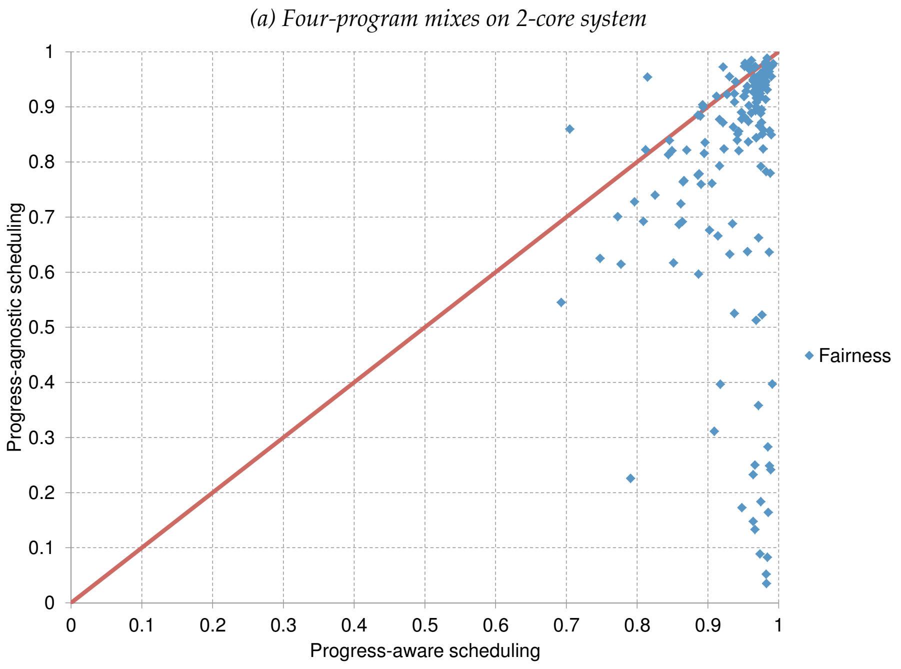
In Chapter 2 we discussed the problem of interference between threads running on a multi-core processor. In Section 6.2.1 we discussed the design of a profiling tool for speedup stacks, and we proposed a counter architecture that captures the impact of resource sharing on the performance of threads. We now use this counter architecture to improve multi-core scheduling.

Progress-aware scheduling leverages the counter architecture to track per-thread progress and schedules slowly-progressing threads more frequently so that they are able to catch up and achieve better performance. Progress-agnostic scheduling assumes that each thread makes equal progress during each timeslice, however, a thread that suffers more from resource contention will observe a higher slowdown compared to other threads. The pitfall is that the scheduler in an OS or VMM is unaware of this slowdown, which may lead to severely degraded performance for workloads that suffer significantly from resource contention.

To evaluate thread-progress aware scheduling, we set up the following experiment. We consider 4-program and 8-program workload mixes, which we schedule on a 2-core and 4-core system, respectively. We assume a 5ms timeslice in these experiments, and we simulate until at least one benchmark has executed 1 billion instructions. The scheduling techniques are implemented in the simulator itself, as we do not simulate the operating system in these experiments. We use the the same experimental setup as in Section 6.2.1.

The baseline progress-agnostic scheduling policy schedules programs such that they all get an equal amount of timeslices; e.g., round-robin achieves this property. Progress-aware scheduling, on the other hand, tracks progress for each of the programs in the mix, and prioritizes the programs with the highest current slowdown to be scheduled first. Slowdown is computed as the execution time on the multi-core system divided by the estimated isolated execution time. In other words, progress-aware scheduling aims at speeding up slow-progress programs so that all users experience good performance and none of the programs experience huge slowdowns nor starvation.

Figure 7.1 reports the progress-agnostic and progress aware scheduling fairness observed for each of the workload mixes for the 2-core and 4-core systems. Fairness [22] is defined as the progress of the slowest



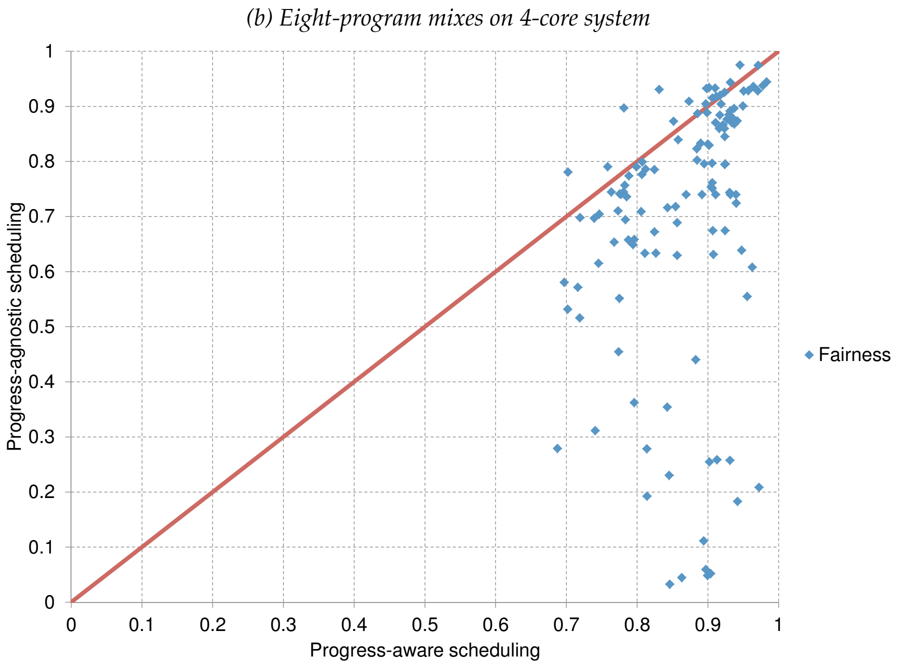


**Figure 7.1:** Fairness results for progress-aware and progress-agnostic scheduling for (a) 4-program mixes on 2 cores, and (b) 8-program mixes on 4 cores.

program divided by the progress of the fastest program in the job mix. A fairness of 1 means that each thread has made the same progress, while a zero fairness means that at least one thread is starving. Each point represents the progress-agnostic (Y-axis) and progress-aware (X-axis) scheduling fairness of a specific workload. Points on the indicated bisector have the same fairness for both scheduling policies. Points beneath the bisector have higher fairness for the progress-aware scheduling, while points above the bisector show a smaller fairness for progress-aware scheduling compared to progress-agnostic scheduling.

Progress-aware scheduling improves fairness substantially over progress-agnostic scheduling for most of the workloads: the majority of the points are located beneath the bisector. For the points at the bottom right corner, the fairness improvement is the largest: a smaller than 0.1 fairness for the progress-agnostic scheduling becomes a bigger than 0.9 fairness for the progress-aware scheduling, which uses the proposed cycle accounting architecture. We observe an average fairness improvement of 20.3% and 24.8% for 2 and 4 cores, respectively.

Improving fairness for threads running on a multi-core processor is important for avoiding missed deadlines in soft real-time applications, for reducing jitter in the response time for interactive applications, for guaranteeing fairness in consolidated environments, for delivering service-level agreements, for balanced performance in parallel workloads, etc.

## 7.3 Improving Multi-Threaded Program Performance

In the previous section, we optimized the fairness between threads from a multi-programmed workload by improving the scheduler. In this section, we improve the performance of a multi-threaded application. To achieve this, we use our criticality metric combined with a dynamic algorithm that accelerates a critical thread during the execution using frequency scaling. The algorithm dynamically measures thread criticality over a timeslice, and scales up the identified most critical thread in the next timeslice. While we evaluate frequency scaling on only one thread, scaling multiple threads could be an option if criticality stacks reveal that this could be worth the energy cost. This dynamic optimization requires no offline analysis, reacts to phase behavior, and improves parallel program execution time.

We first detail our dynamic algorithm, then compare results of our dynamic approach with those gathered for an offline approach. For the offline approach, we run the program twice, one iteration for identifying the critical thread and then a second iteration where we accelerate this thread during the whole execution of the application. We also compare our dynamic approach with prior work called BIS, showing we almost double their performance improvements. In Section 7.4, we will show that this dynamic algorithm also leads to a more energy-efficient execution of our parallel applications.

Exploring a large frequency range in Chapter 4 showed that using a 2.5 GHz frequency achieves the largest speedups relative to the amount of scaling, while not overly consuming energy. Hence, in these experiments, we raise a critical thread's frequency to 2.5 GHz. We further assume a multi-core processor with a base frequency of 2 GHz, where the processor's Thermal Design Power (TDP) allows one and only one core's frequency to be increased. We use a timeslice of 10 ms for our dynamic algorithm. At the start of a new timeslice, we reset criticality counters. Over the timeslice, the hardware calculates each thread's criticality sum using the hardware component for measuring thread criticality (see Section 4.2.2). Algorithm 1 details how these criticality numbers are used to decide which (if any) core to scale up in the next timeslice.

**Algorithm description.** Initially, we check if there is currently an accelerated core, tracked with $f$. Calculated criticality numbers are stored in $C_i$ for each thread. If no core is accelerated ($f = \emptyset$), we calculate the ratio between the largest and the smallest criticality. If the result is larger than a certain threshold $\alpha$ (the base value is 1.2), then the frequency of the core running the thread with the largest criticality component is raised (by setting $f$ to the index of the core with maximum $C_i$).

If a core was accelerated in the previous slice, we check the ratio of the largest criticality to the smallest criticality that is not the currently accelerated core (taking the second-smallest criticality if the smallest is for the accelerated thread). We perform this check to prevent constantly scaling up and down a core, since speeding up a thread will usually result in a smaller criticality component. If this ratio is above our $\alpha$ threshold, we raise the frequency of the core running the most critical

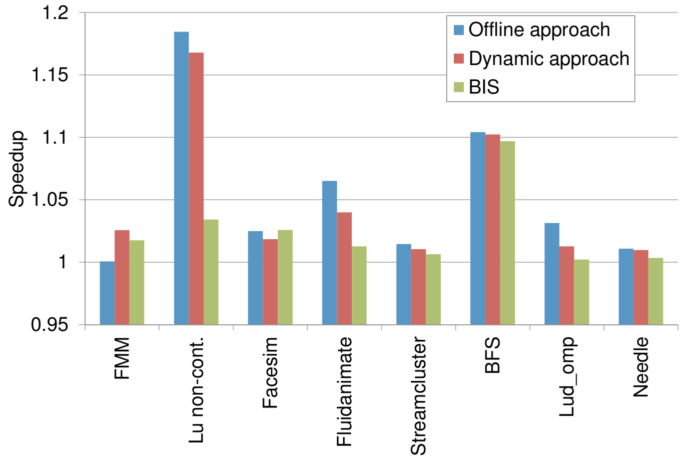
```
if $f = \emptyset$ then
    if $\max(C_i)/\min(C_i) > \alpha$ then
        $f$ :=maxindex($C_i$)
    end
else
    if $\max(C_i)/\min_{i \neq f}(C_i) > \alpha$ then
        $f$ :=maxindex($C_i$)
    else
        if $C_f/\max(C_i) < \beta$ then
            $f := \emptyset$
        end
    end
end
```

**Algorithm 1:** Dynamic frequency scaling algorithm. $f$ is the currently accelerated core; $C_i$ is the criticality for thread $i$; 'maxindex' finds the index of the core with maximum criticality.

thread (slowing down the previously accelerated thread if it is different). If the ratio is not larger than the threshold, the algorithm calculates the ratio of the criticality of the thread running on the accelerated core to the largest criticality. If this ratio is smaller than a $\beta$ threshold (with a base value of 0.8), then the accelerated thread is slowed down again. This check prevents continuously accelerating a core without seeing a performance benefit, as a thread that was initially critical can eventually become non-critical. We performed experiments in which we vary timeslice duration, and the $\alpha$ and $\beta$ parameters, but found little performance difference as compared to using the base values.

In addition to this proactive algorithm, we implemented two additional straightforward reactive mechanisms to further reduce energy consumption and improve performance. First, when an accelerated thread is scheduled out by the OS, we reduce the frequency of that core to the base frequency, as speeding up that thread has no performance benefit. Secondly, when there is only one thread active, and that thread is currently not accelerated, we scale up the frequency of the core running that thread. In this case, the running thread is by definition the most critical thread, and should be accelerated immediately, without waiting for the next timeslice.

*(a) 8 threads*



*(b) 16 threads*

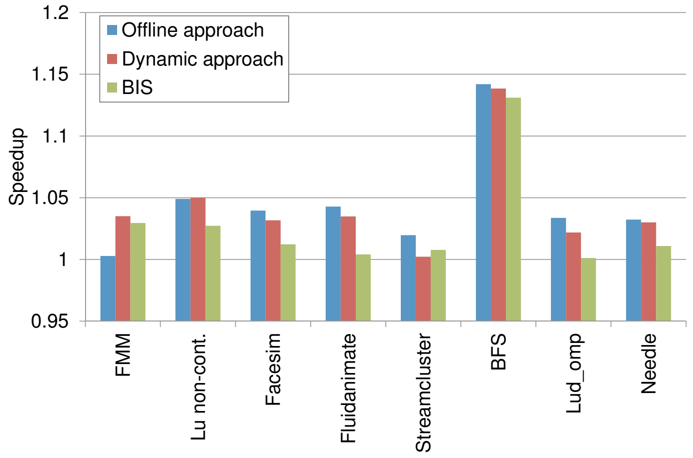


**Figure 7.2:** Results for the dynamic frequency scaling policy.

### 7.3.1 Effectiveness of Dynamic Optimization

Figure 7.2 shows the performance results of our dynamic frequency scaling technique for both 8 and 16-threaded configurations. Because FMM's total program criticality stack did reveal a most critical thread, we include it again in our dynamic results, despite the fact that speeding up one thread over the whole execution did not improve performance. Each benchmark has three bars. The first bar is the speedup obtained by the offline approach, i.e., profiling the program and running the program again while speeding up the most critical thread over the whole program execution. The next bar shows the speedup obtained by our dynamic approach. The last bar shows the results for BIS, which we discuss in the next section.

For both 8 and 16-threaded runs, FMM achieves larger speedups with our dynamic approach than the offline approach. Although the offline approach could not improve FMM's performance, our dynamic approach deals better with the overlapping criticality, and improved performance by about 3%. Also, as discussed in the previous chapter, the offline approach could not solve 16-threaded Lu non-cont.'s problem that one thread was most critical initially and another was critical later in the program. The dynamic approach slightly improves upon the performance of Lu non-cont. with 16 threads, adapting to the most critical thread during each program phase.

For the other benchmarks, the speedups of the dynamic approach are slightly smaller than those of the offline approach. This is due to the reactiveness of the dynamic algorithm: frequency is only scaled up after a critical thread is detected in the previous timeslice. However, the dynamic approach achieves similar program speedups with more a energy-efficient run by not always scaling up the frequency. On average, the dynamic approach adapts to phase behavior, obtaining a speedup of 4.4%, compared to 4.8% for the offline approach, while speeding up one thread from 2 GHz to 2.5 GHz for 71% of the time on average.

### 7.3.2 Comparison to Previous Work

We compare the results of our dynamic frequency scaling algorithm to the best-performing previous work which accelerates synchronization bottlenecks instead of threads, called Bottleneck Identification and Scheduling (BIS) [34]. They focus on accelerating the most critical bottleneck, e.g., a critical section that is heavily contended or a barrier with many threads waiting for a significant amount of time. When a thread encounters such a bottleneck, it is temporarily migrated to a faster core in a heterogeneous system. We reimplemented their technique but instead of thread migration, we use core frequency scaling (to 2.5 GHz) in our experimental setup.

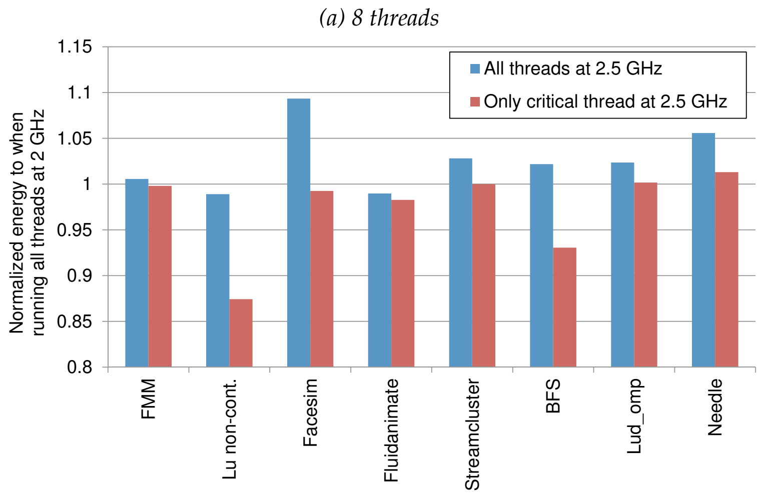
Figure 7.2 presents the speedup for each benchmark using our dynamic algorithm against those obtained using the BIS technique. For 8-threaded benchmarks, in Figure 7.2(a), we see our criticality metric outperforms BIS in all but one benchmark, significantly outperforming BIS for Lu non-cont. by speeding up the benchmark 17% compared to 3% for BIS. Similarly, our dynamic algorithm improves upon BIS's speedup in all 16-threaded benchmarks except for Streamcluster. We found that our technique is more effective at speeding up programs that have many barriers, because we speed up more of the whole thread's execution instead of only when a single thread that has yet to reach the barrier. For programs with many heavily contending critical sections, BIS might achieve better performance. Overall, our dynamic scheme achieves an average of 4.6% speedup in comparison with BIS's 2.4% for 8 threads. For 16 threads, we speed up on average by 4.2%, almost doubling BIS's improvement of 2.7%.

## 7.4 Improving Multi-Threaded Program Energy Usage

While we wanted to achieve maximum performance for parallel programs in the previous section, power and energy are first-order concerns in modern systems, including the embedded and server domains. For optimizing towards this dimension we again use our criticality stacks.

We perform an experiment to compare the energy consumed when running our multi-threaded benchmarks at various frequencies. We run once with all threads at 2 GHz, once with all threads at 2.5 GHz, and once using our dynamic technique to accelerate only the most critical thread to 2.5 GHz. Obviously, running all threads at the higher frequency will result in a larger power output. Figure 7.3 presents the energy consumed, which is power multiplied by execution time, for our benchmarks with 8 and 16 threads. We present energy numbers for all threads at 2.5 GHz, and only the critical thread at 2.5 GHz, normalized to the energy consumption for all threads at 2 GHz. We estimate power consumption using McPAT [41] (assuming a 32 nm technology).

Figure 7.3 shows that accelerating all threads to the higher frequency consumes more energy than accelerating only one thread for all of our benchmarks. For both Lu non-cont. and Fluidanimate, running with all threads at 2.5 GHz consumes slightly less energy than with all threads at 2 GHz, because it results in large program speedups. However, if energy is of prime concern, we see the best result comes



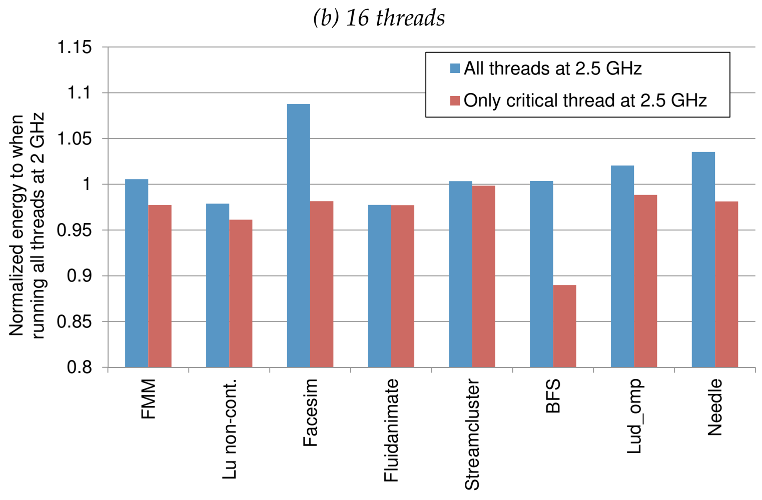


**Figure 7.3:** Comparison of energy consumed when running all threads at 2.5 GHz and only the most critical at 2.5 GHz using our dynamic scheme, compared to running all threads at 2 GHz.

from targeting acceleration only at the most critical thread. For almost all benchmarks, using our dynamic algorithm reduces the energy consumed from all threads at 2 GHz. Particularly for BFS with 16 threads, and Lu non-cont. with 8 threads, we reduce the energy consumed by 11% and 12.6%, respectively. Also, targeting acceleration to the thread identified as most critical by our metric particularly benefits Facesim, which consumes about 10% more energy when all threads are accelerated. Overall, when all threads are executed at 2.5 GHz, the total energy consumption *increases* by 1.3% for 16 threads and 2.5% for 8 threads. In comparison, by accelerating only the critical thread, the total energy consumption is *reduced* by 3.2% on average for 16 threads and 2.8% for 8 threads.

## 7.5 Related Work

Improving parallel performance by reducing thread waiting time is a well-known optimization paradigm. Many previously proposed mechanisms apply this conventional wisdom for specific performance idioms.

Threads wait for several reasons. The most obvious case is serial execution parts of a parallel program [2]. When there is only one thread active doing useful work, optimizing its performance is likely to yield substantial performance benefits. Annavaram et al. [3] optimize serial code by running at a higher clock frequency; Morad et al. [49] run serial code on a big core in a heterogeneous multi-core.

Critical sections guarantee mutual exclusion and lead to serialization, which puts a fundamental limit on parallel performance [23]. Removing or alleviating serialization because of critical sections has been a topic of wide interest for many years. Transactional Memory (TM) aims to overlap the execution of critical sections as long as they do not modify shared data [30]. Speculative Lock Elision [53], Transactional Lock Removal [54] and Speculative Synchronization [46] apply similar principles to traditional lock-synchronized programs. Suleman et al. [59] use the big core in a heterogeneous multi-core to accelerate critical sections.

Several techniques have been proposed to improve performance and/or reduce energy consumption of barriers, which all threads have to reach before the program proceeds. In thrifty barriers [40], a core is put into a low-power mode when it reaches a barrier with a predicted long stall time. Liu et al. [44] improve on that by reducing the frequency of cores running threads that are predicted to reach a barrier much sooner than other threads, even when they are still executing. Cai et al. [11] keep track of how many iterations of a parallel loop each thread has executed, delaying those that have completed more, and giving more resources to those with fewer in an SMT context. Age-based scheduling [39] uses history from the previous instance of the loop to choose the best candidate for acceleration. While previous works all target a specific synchronization paradigm (barriers and parallel loops), our criticality metric is independent of the type of synchronization, and can profile every (instrumented) stall event due to synchronization.

Turbo Boost[1] increases the core frequency when there are few active cores. As such, for multi-threaded programs, it increases thread performance when parallelism is low. Booster [47] speeds up threads that hold locks or that are active when other threads are blocked, using a dual voltage supply technique. Bottleneck Identification and Scheduling (BIS) by Joao et al. [34] accelerates synchronization primitives (locks, barriers, pipes) with large amounts of contention by migrating them temporarily to a faster core in a heterogeneous multi-core. The methods used by both Turbo Boost and Booster to identify threads that need to be accelerated are a subset of the methods used by BIS, which means that the BIS results in Section 7.3.2 are an upper bound for the results for Turbo Boost and Booster. While BIS optimizes bottlenecks, we identify the thread(s) most critical to overall performance. Optimizing bottlenecks does not necessarily imply improved overall performance, because they also accelerate non-critical threads. In Section 7.3.2, we showed that our dynamic algorithm results in a higher speedup than BIS for barrier-bound applications.

## 7.6 Summary

In this chapter we optimized the performance of applications in three dimensions. We first showed a mechanism to improve fairness between threads running on a multi-core processor. By making the scheduler aware of the progress of threads, and scheduling slowly-progressing threads more frequently, we report an average fairness improvement of 22.5% over progress-agnostic scheduling. We then proposed an algorithm to dynamically optimize the performance of multi-threaded applications, achieving an average speedup of 4.4%, and up to 17%. Finally we showed how the energy consumption of multi-threaded applications can be reduced by accelerating the most critical thread, resulting in a reduction of the total energy consumption by 3% on average and up to 12.6%.

[1]http://www.intel.com/technology/turboboost

# Chapter 8

# Conclusions and Future Work

*In this chapter we summarize the conclusions from this dissertation and elaborate on potential avenues for future work.*

## 8.1 Summary

Threads executing on a multi-core processor affect each other's performance at two levels. First, there is interference between the cores due to hardware resource sharing on modern multi-core processors, in which co-executing threads compete for shared resources, such as caches, on-chip interconnection network, off-chip bandwidth, memory banks, etc. For multi-threaded applications, this interference can either have a positive or a negative impact on performance. An example of negative interference occurs in shared caches where threads can evict data of other threads, which means that the latter will experience additional misses compared to isolated execution. On the other hand, cache sharing can also lead to positive interference when threads load data into the cache that can later be used by other threads.

Secondly, next to resource sharing, threads of a multi-threaded application also interfere with each other due to synchronization between the threads. Typical examples of synchronization primitives are barriers, critical sections and consumer-producer relationships. While synchronization is necessary to achieve a correct execution of a parallel program, it leads to threads waiting for each other, either in a spinning or yielding state.

Those two interactions between threads make it very challenging to analyze the performance and scalability of multi-threaded applications. However, analyzing parallel performance and identifying scaling bottlenecks is key to optimize both multi-threaded software and hardware. We therefore propose three new performance analysis methods in this dissertation, namely *criticality stacks, bottle graphs* and *speedup stacks*.

In this dissertation we introduce a novel, intuitive criticality metric that is independent of synchronization primitives, and that takes into account both a thread's active running time and the number of threads waiting on it. We use this metric to create *criticality stacks* that break down total execution time visually based on each thread's criticality, facilitating detailed analysis of parallel imbalance. We describe a simple hardware design that takes a very small amount of power, while being off the processor's critical path, to compute criticality stacks during execution. We validate the accuracy and utility of criticality stacks by demonstrating that our low-overhead online calculation approach indeed finds the thread most critical to performance, improving over a previously proposed metric based on cache misses. We also use criticality stacks for various use cases in this dissertation that illustrate their broad applicability to (1) optimize software code, (2) dynamically accelerate the critical thread to improve performance, even doubling over the best-performing previous work, and (3) target optimizations of parallel programs to reduce energy consumption. From these case studies, we report that (1) after optimizing the code of one benchmark based on criticality imbalance, we achieve an average speedup of 1.9×; (2) our dynamic algorithm reacts to application phase changes, achieving an average speedup of 4.4%, and up to 17%; (3) by accelerating the most critical thread, we also reduce the total energy consumption by 3% on average, and up to 12.6% (while at the same time improving performance). Overall, we conclude that criticality stacks are instrumental for analyzing parallel program thread imbalance due to synchronization, and guiding online optimizations to improve performance and/or reduce energy consumption of multi-threaded applications on multi-core processors.

Besides constructing criticality stacks, we also produce *bottle graphs* with our new criticality metric. Bottle graphs are an intuitive and useful method for visualizing multi-threaded application performance, analyzing scalability bottlenecks on a per-thread level, and targeting optimization. Bottle graphs represent each thread as a box, showing its execution time share (height), parallelism (width), and total running time (area). The total height of the bottle graph when these boxes are stacked on top of each other is the total application execution time. Because we place threads with the largest parallelism on the bottom, the neck of the bottle graph points to threads that offer the most potential for optimization, i.e., those with limited parallelism and a large execution time share. Although bottle graphs can be constructed with the hardware component that we designed for creating criticality stacks, we designed an implementation in software that uses light-weight OS modules to calculate the components of bottle graphs for unmodified parallel programs running on real hardware, at very low overhead (0.68% on average). We use bottle graphs in this work to analyze a set of 12 Java benchmarks, revealing scalability limitations in several well-known applications and suggesting optimizations. We use our bottle graphs to analyze Jikes' RVM service threads, revealing very limited parallelism when increasing the number of garbage collection threads beyond two due to extra synchronization activity. We have compared this to OpenJDK's garbage collector which scales much better for our thread counts. Bottle graphs are a powerful visualization method that is necessary for tackling multi-threaded application bottlenecks in modern multi-core hardware.

We also propose *speedup stacks*, that visualize the achieved speedup of an application and the various scaling delimiters as a stacked bar. The height of a speedup stack equals the number of threads, and the stack components denote the contributions of the scaling delimiters. The intuition behind speedup stacks is that the relative importance of the scaling delimiters is immediately clear from the speedup stack, hence, it is an insightful method for driving both hardware and software optimization. The concept of a speedup stack is applicable to a broad range of multi-threaded, multi-core and multi-processor systems. We propose two versions of speedup stacks in this work. Our first version uses additional hardware support for constructing speedup stacks. In this version of speedup stacks we include LLC and memory subsystem interference, spinning, yielding, and thread imbalance as scaling delimiters. For generating these speedup stacks, we designed a dedicated counter architecture for measuring the impact of resource sharing on the performance of threads. This cycle accounting architecture estimates the impact of negative interference due to inter-thread misses in the shared cache as well as the resource and bandwidth sharing in the memory subsystem, including the memory bus, bank conflicts and row buffer conflicts. Besides negative interference the counter architecture also estimates the impact of positive interference between threads on performance. The hardware cost is limited to 1.1 KB per core or a total of 18 KB for a 16-core CMP. The accuracy of these speedup stacks is within 5.1% average absolute error (error is defined as the difference between estimated and measured speedup for an application) across a broad set of SPLASH-2, PARSEC and Rodinia benchmarks. We demonstrate the usage of these speedup stacks for identifying scaling bottlenecks, for classifying benchmarks based on their scaling delimiters, and for understanding LLC performance. Our second version of speedup stacks is targeted towards managed language applications and uses an extended version of the software profiling tool that we designed for generating bottle graphs. This way we can generate speedup stacks for applications running on native hardware. In this version of speedup stacks, we include a different set of scaling delimiters, namely garbage collection, sequential parts, thread imbalance, synchronization between threads and hardware interference. We use this second version of speedup stacks to understand the scaling behavior of Java applications running on Jikes RVM with a stop-the-world and a concurrent garbage collector.

## 8.2 Future Work

In this dissertation we proposed new methods for analyzing and optimizing the performance of multi-threaded applications, but there is still room for future work which we will discuss in the following sections.

### Performance Analysis of Distributed Applications

While we analyzed the performance of multi-threaded applications running on one physical machine, there also exist applications running with multiple threads on different machines, for example MPI applications from the HPC domain. If we want to analyze this type of program, we have to include a third dimension, besides resource sharing and synchronization, which is the communication link between the different machines. Just like synchronization this communication link can cause threads to wait and consequently have an impact on the execution time of the program. This can also apply to other managed languages that use communication channels, like Erlang.

Therefore, if we want to analyze these applications we have to extend our criticality metric so that it includes waiting time due to communication between the machines. We also need to design a new profiling tool that calculates criticality across different machines.

### Performance Analysis of Multi-Tier Applications

A second type of application we did not analyze in this dissertation are programs that contain multiple layers (hence the name multi-tier) and where each layer is organized as a thread pool that does work. We can analyze these applications with bottle graphs, where each bottle graph represents the threads of a thread pool from one layer. The bottle graphs then reveal for each thread pool if there is work (im)balance between the threads and how many threads are active on average in the thread pool. However, the reason why parallelism of threads is limited inside a thread pool might be because there are too few requests coming from the upper layer. Therefore, it makes sense to extend the bottle graphs to a hierarchical representation that incorporates the behavior of all layers into one graph.

### Improved Scheduling on a Heterogeneous Multi-Core Processor

Heterogeneous multi-core processors are processors where not all the cores have the same architecture, typically containing a mix of fast and big cores with slow and small cores. Scheduling multi-threaded workloads on this type of processor is very challenging because not all the cores have the same performance. There has been research on this topic in the past [34, 59, 62], and all of this prior work either tries to identify bottlenecks in the application (for example heavily contended locks) or uses architectural characteristics to steer thread scheduling. With our criticality metric we can identify the threads that are most determinative of execution time, and use this to steer scheduling. The dynamic optimization algorithm from Chapter 7 can decide which threads should be executed on the big core. If there are no critical threads, we can do a round robin scheduling of threads on the big core.

### Improving the Performance of Jikes RVM

While we extensively evaluated the performance of Jikes and the applications running on top of it in Chapter 5 and 6, we did not optimize the performance. For example, we noticed that Jikes' garbage collector does not scale very well, but we did not look into the causes of this poor scaling. This could be an interesting direction for future research. Improving the performance of the stop-the-world garbage collector inside a JVM will definitely lead to a performance gain for the applications running on top of it.

Secondly, the information from the bottle graphs can also be used by the JVM to trigger code optimization for certain threads. For example if the bottle graphs reveal that some threads are more critical to performance than others, then this information can be used by the JVM to optimize the source code of these critical threads more aggressively, which would lead to a performance gain for the application.

# Bibliography

[1] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance Analysis of Idle Programs. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 739–753, Oct. 2010.

[2] G. M. Amdahl. Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities. In *Proceedings of the American Federation of Information Processing Societies Conference (AFIPS)*, pages 483–485, 1967.

[3] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law through EPI Throttling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 298–309, June 2005.

[4] A. Bhattacharjee and M. Martonosi. Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 290–301, June 2009.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, Oct. 2008.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 Simulator. *Computer Architecture News*, 39:1–7, May 2011.

[7] S. M. Blackburn and A. L. Hosking. Barriers: Friend or Foe? In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 143–151, Oct. 2004.

[8] S. M. Blackburn and K. S. McKinley. Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *Proceedings of the Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 22–32, June 2008.

[9] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 169–190, Oct. 2006.

[10] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 83–94, June 2000.

[11] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting Points: Using Thread Criticality to Adapt Multicore Hardware to Parallel Regions . In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 240–249, Sept. 2008.

[12] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The Yin and Yang of Power and Performance for Asymmetric Hardware and Managed Software. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 225–236, June 2012.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct. 2009.

[14] G. Chen and P. Stenström. Critical Lock Analysis: Diagnosing Critical Section Bottlenecks in Multithreaded Applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 71:1–71:11, Nov. 2012.

[15] J. Demme and S. Sethumadhavan. Rapid Identication of Architectural Bottlenecks via Precise Event Counting. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 353–364, June 2011.

[16] K. Du Bois, T. Schaeps, S. Polfliet, F. Ryckbosch, and L. Eeckhout. SWEEP: Evaluating Computer System Energy Efficiency Using Synthetic Workloads. In *Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, pages 159–166, Jan. 2011.

[17] K. Du Bois, S. Eyerman, and L. Eeckhout. Per-thread Cycle Accounting in Multicore Processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–22, Jan. 2013.

[18] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout. Criticality Stacks: Identifying Critical Threads in Parallel Programs Using Synchronization Behavior. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 511–522, June 2013.

[19] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-threaded Applications. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming, Systems Languages and Applications (OOPSLA)*, pages 355–372, Oct. 2013.

[20] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout. Analyzing Scaling Behavior of Managed Runtime Applications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2014. Under review.

[21] P. G. Emma. Understanding Some Simple Processor-Performance Limits. *IBM Journal of Research and Development*, 41(3):215–232, May 1997.

[22] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multi-Program Workloads. *IEEE Micro*, 28(3):42–53, May/June 2008.

[23] S. Eyerman and L. Eeckhout. Modeling Critical Sections in Amdahl's Law and its Implications for Multicore Design. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 362–370, June 2010.

[24] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A Performance Counter Architecture for Computing Accurate CPI Components. In *Proceedings of The Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–184, Oct. 2006.

[25] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A Mechanistic Performance Model for Superscalar Out-of-Order Processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2):42–53, May 2009.

[26] S. Eyerman, K. Du Bois, and L. Eeckhout. Speedup Stacks: Identifying Scaling Bottlenecks in Multi-Threaded Applications. In *Proceedings of the International Symposium on Performance Analysis of Software and Systems (ISPASS)*, pages 145–155, Apr. 2012.

[27] B. Fields, S. Rubin, and R. Bodík. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 74–85, June 2001.

[28] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: Rethinking and Rebooting gprof for the Multicore Age. In *Proceedings of the Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 458–469, June 2011.

[29] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A Study of the Scalability of Stop-The-World Garbage Collectors on Multicores. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 229–240, Mar. 2013.

[30] M. Herlihy and J. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 289–300, June 1993.

[31] J. Hollingsworth. An Online Computation of Critical Path Profiling. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 11–20, 1996.

[32] Intel. Intel VTune$^{TM}$ Amplifier XE 2011. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[33] M. Itzkowitz and Y. Maruyama. HPC Profiling with the Sun Studio$^{TM}$ Performance Tools. In *Tools for High Performance Computing 2009*, pages 67–93. Springer, 2010.

[34] J. Joao, M. Suleman, O. Mutlu, and Y. Patt. Bottleneck Identification and Scheduling in Multithreaded Applications . In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–234, Mar. 2012.

[35] L. K. John. More on Finding a Single Number to Indicate Overall Performance of a Benchmark Suite. *ACM SIGARCH Computer Architecture News*, 32(4):1–14, Sept. 2004.

[36] T. Kalibera, M. Mole, R. Jones, and J. Vitek. A Black-Box Approach to Understanding Concurrency in DaCapo. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 335–354, Oct. 2012.

[37] M. Kambadur, K. Tang, and M. A. Kim. Harmony: Collection and Analysis of Parallel Block Vectors. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 452–463, June 2012.

[38] R. E. Kessler, M. D. Hill, and D. A. Wood. A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches. *IEEE Transactions on Computers*, 43(6):664–675, June 1994.

[39] N. B. Lakshminarayana, J. Lee, and H. Kim. Age Based Scheduling for Asymmetric Multiprocessors. In *Proceedings of Supercomputing: the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 199–210, Nov. 2009.

[40] J. Li, J. Martinez, and M. Huang. The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 14–23, Feb. 2004.

[41] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 469–480, Dec. 2009.

[42] T. Li, A. Lebeck, and D. Sorin. Quantifying Instruction Criticality for Shared Memory Multiprocessors. In *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 128–137, 2003.

[43] T. Li, A. R. Lebeck, and D. J. Sorin. Spin Detection Hardware for Improved Management of Multithreaded Systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 17:508–521, June 2006.

[44] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. Irwin. Exploiting Barriers to Optimize Power Consumption of CMPs. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, page 5a, Apr. 2005.

[45] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero. ITCA: Inter-Task Conflict-Aware CPU Accounting for CMPs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 203–213, Sept. 2009.

[46] J. F. Martinez and J. Torrellas. Speculative Synchronization: Applying Thread-level Speculation to Explicitly Parallel Applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 18–29, Oct. 2002.

[47] T. Miller, X. Pan, R. Thomas, N. Sedaghati, and R. Teodorescu. Booster: Reactive Core Acceleration for Mitigating the Effects of Process Variation and Application Imbalance in Low-Voltage Chips. In *18th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Feb. 2012.

[48] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics Magazine*, 38(8):1–6, Apr. 1965.

[49] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and A. Ayguade. Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors. *IEEE Computer Architecture Letters*, 5(1):14–17, Jan. 2006.

[50] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings*

*of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–11, Oct. 1996.

[51] OpenJDK. *OpenJDK (Implementation of the Java SE 6 Specification), Version 1.6*. Oracle, 2006. URL `http://openjdk.java.net/projects/jdk6/`.

[52] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 423–432, Dec. 2006.

[53] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 294–305, Dec. 2001.

[54] R. Rajwar and J. R. Goodman. Transactional Lock-free Execution of Lock-based Programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 5–17, Oct. 2002.

[55] A. G. Saidi, N. L. Binkert, S. K. Reinhardt, and T. Mudge. End-to-end Performance Forecasting: Finding Bottlenecks Before They Happen. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 361–370, June 2009.

[56] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, Oct. 2002.

[57] SPEC. *SPECjbb2005 (Java Server Benchmark), Release 1.07*. Standard Performance Evaluation Corporation, 2006. URL `http://www.spec.org/jbb2005`.

[58] STMicroelectronics. PGProf: Parallel Profiling for Scientists and Efngineers. http://www.pgroup.com/products/pgprof.htm, 2011.

[59] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 253–264, Mar. 2009.

[60] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam. Dynamic Recognition of Synchronization Operations for Improved Data Race Detection. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 143–154, July 2008.

[61] E. Tune, D. Liang, D. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 185–195, Feb. 2001.

[62] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 213–224, June 2012.

[63] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 24–36, June 1995.

[64] X. Zhou, W. Chen, and W. Zheng. Cache Sharing Management for Performance Fairness in Chip Multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 384–393, Sept. 2009.