Snelle-simulatietechnieken voor de exploratie van
de microarchitecturale ontwerpruimte

Fast Simulation Techniques for
Microprocessor Design Space Exploration

Davy Genbrugge

Promotor: prof. dr. ir. L. Eeckhout
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
Academiejaar 2009 - 2010

UNIVERSITEIT
GENT

ISBN 978-90-8578-328-2
NUR 980
Wettelijk depot: D/2010/10.500/4

For my family, for my friends.

Dankwoord

Bij de aanvang van dit onderzoek had ik slechts een vaag idee over het pad dat ik zou bewandelen. Al snel had ik door dat onderzoek niet over een rechte geplaveide weg loopt, maar op een kronkelig pad vol obstakels. Meerdere malen ben ik mezelf tegengekomen. Gelukkig kon ik rekenen op de steun van een aantal mensen, die onmisbaar waren bij het tot stand komen van dit werk. Graag wil ik hen uitvoerig bedanken.

In de eerste plaats wil ik mijn promotor prof. Lieven Eeckhout bedanken. Verscheidene malen heb ik me kunnen beroepen op de expertise van Lieven wanneer ik door de bomen het bos niet meer zag. Hij heeft me (met het nodige geduld) de knepen van het vak geleerd, en kon steeds weer begrip opbrengen wanneer ik in de fout ging. Ook op momenten dat de moed me in de schoenen zonk wist hij me te motiveren om verder te gaan. Zonder zijn hulp had ik nooit de tal van mooie ervaringen kunnen opdoen en had ik niet gestaan waar ik nu sta.

In de tweede plaats wil ik prof. Koen De Bosschere bedanken. Reeds meer dan dertien jaar bouwt Koen een onderzoeksgroep uit die jonge mensen de kans geeft zich te verdiepen in computerarchitectuur. Door zijn enthousiasme en gedrevenheid weet hij steeds opnieuw tal van mensen te overtuigen om een onderzoek aan te vatten. Jaren terug heb ik ervoor gekozen om me te vervoegen bij de groep van Koen; iets wat ik mij nog nooit heb beklaagd.

I also would like to thank all members of my PhD committee for their effort to evaluate this thesis. I would like to thank prof. Brad Calder from Microsoft and prof. Erik Hagersten from Acumem and Uppsala University in Sweden. I really appreciate it that you have found the time to read my thesis and to serve on my committee; I know that you have a tight schedule. Your comments were invaluable for improving my thesis. Verder wil ik ook de binnenlandse leden van mijn doctoraatscommissie bedanken voor hun inspanning om dit werk te beoordelen: prof. Diederik Verkest, prof.

Chris Develder, prof. Koen De Bosschere en prof. Jan Van Campenhout. Uw suggesties waren van onschatbare waarde om mijn thesis sterker te maken. Uiteraard wil ik ook prof. Rik Van de Walle bedanken om de taak van voorzitter op zich te nemen.

Jarenlang heb ik kunnen werken in een aangename collegiale werksfeer. In de eerste plaats bedank ik mijn bureaugenoten: Andy, Dries, Kris, Frederick, Stijn en Max. Het was/is een plezier om met jullie een bureau te delen. In de afgelopen jaren heb ik ontzettend veel bijgeleerd ondermeer van mijn collega's: Michiel, Stijn, Andy, Jonas, enz. Ongetwijfeld ben ik een aantal mensen vergeten, daarom wil ik *iedereen* bedanken voor die mooie jaren. In het bijzonder wil ik Andy en Stijn nog eens bedanken om mijn thesis na te lezen.

Gedurende mijn onderzoeksactiviteiten was ik tewerkgesteld op een aantal projecten. Bij deze wens ik dan ook het Instituut voor de aanmoediging van Innovatie door Wetenschap en Technologie in Vlaanderen (IWT), het Fonds voor Wetenschappelijk Onderzoek Vlaanderen (FWO) en de Universiteit Gent bedanken voor de financiering van dit onderzoek.

Tot slot wens ik ook mijn familie en vrienden te bedanken om mij al die jaren te steunen. Ik was niet altijd het zonnetje in huis, zeker niet op moment dat er een *deadline* moest gehaald worden. Vaak heb ik tegen mijn kameraden 'nee' moeten zeggen wegens 'geen tijd'. Daarom bedankt voor al jullie begrip! In het bijzonder, mama, papa en zus (en Pieter), jullie zijn zoals altijd de beste! Genoeg reden voor mij om te blijven doorgaan, dank u.

Davy Genbrugge
Gent, 22 januari 2010

Examencommissie

Prof. Rik Van de Walle, voorzitter
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Jan Van Campenhout, secretaris
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

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

Prof. Brad Calder
Department of Computer Science and Engineering
University of California in San Diego
Microsoft
USA

Prof. Koen De Bosschere
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Chris Develder
Vakgroep INTEC, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Erik Hagersten
Department of Information Technology
Uppsala University
Acumem
Sweden

Prof. Diederik Verkest
Vakgroep ETRO, Faculteit Ingenieurswetenschappen
Vrije Universiteit Brussel
IMEC Leuven

Leescommissie

Prof. Brad Calder
Department of Computer Science and Engineering
University of California in San Diego
Microsoft
USA

Prof. Chris Develder
Vakgroep INTEC, Faculteit Ingenieurswetenschappen
Universiteit Gent

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

Prof. Erik Hagersten
Department of Information Technology
Uppsala University
Acumem
Sweden

Prof. Diederik Verkest
Vakgroep ETRO, Faculteit Ingenieurswetenschappen
Vrije Universiteit Brussel
IMEC Leuven

Samenvatting

Het ontwerpen van een nieuwe microprocessor is enorm tijdrovend: het kan tot zeven jaar duren alvorens een processor op de markt komt. Processorontwerpers maken gebruik van cyclusgetrouwe simulatoren tijdens de verschillende stadia van het ontwerpproces, bijvoorbeeld bij het nemen van hoog-niveaubeslissingen tijdens de initiële verkenning van de ontwerpruimte. Architecturale simulatoren modelleren de microarchitectuur in software. Deze simulatoren zijn zeer nauwkeurig en zeer flexibel in gebruik, maar hebben als nadeel dat ze drie tot vier grootteordes trager zijn dan de hardware die ze modelleren. Bovendien vraagt het veel tijd om ze te ontwikkelen.

De huidige trendverschuiving naar chip-multiprocessors maakt deze laatstgenoemde problemen erger. Chip-multiprocessors combineren meerdere kernen op één enkele chip, en delen een aantal systeemhulpbronnen zoals caches, geheugenbussen, hoofdgeheugen, enz. Draden die gelijktijdig worden uitgevoerd kunnen elkaars prestatie beïnvloeden via deze gedeelde componenten. Bijvoorbeeld, een draad kan trager vooruitgang boeken dan een andere draad ten gevolge van bijkomend conflictgedrag. Bovendien kan een verandering in de microarchitectuur ervoor zorgen dat andere delen van de draden op hetzelfde moment in uitvoering zijn, wat op zijn beurt leidt tot een verschillend conflictgedrag en dus een verschillende relatieve vooruitgang van de draden. Deze verstrengeling van prestatie tussen de draden in uitvoering maakt het moeilijk om de prestatie van chip-multiprocessors te modelleren. Naarmate het aantal kernen op een processor toeneemt, neemt de simulatiesnelheid aan belang toe.

Onderzoekers en computerarchitecten zijn zich bewust van het simulatieprobleem en hebben reeds tal van technieken voorgesteld om de simulatie te versnellen. In dit proefschrift stellen we twee technieken voor die de simulatiesnelheid aanzienlijk verbeteren, namelijk, statistische simulatie en intervalsimulatie.

Statistische simulatie versnelt de simulatie door het aantal gesimuleerde instructies te verminderen. Dit gebeurt in drie stappen. Eerst meten we een aantal programmakarakteristieken op in een zogenaamd statistisch profiel door middel van functionele simulatie of andere profileringshulpmiddelen. Op basis van dit profiel wordt dan een synthetische trace gegenereerd met dezelfde eigenschappen, maar bestaande uit veel minder instructies – in de orde van enkele miljoenen instructies. Door de synthetische trace te simuleren verkrijgen we snel een schatting van de prestatie.

We leveren twee belangrijke bijdragen aan het statistische simulatieparadigma. Ten eerste verbeteren we de nauwkeurigheid door een betere modellering van de geheugendatastromen; dit houdt onder andere in dat we geheugenparallellisme, secundaire cachemissers, en load-store data-afhankelijkheden modelleren. Ten tweede breiden we het statistische simulatieparadigma uit om het conflictgedrag in gedeelde systeemhulpbronnen van chip-multiprocessors die meerdere programma's uitvoeren te kunnen vatten. Dit vereist dat we toegangen tot het geheugensubsysteem onafhankelijk van de microarchitectuur modelleren; hiervoor gebruiken we metrieken zoals hergebruiksafstand en stapeldiepte. Een bijkomend voordeel van een microarchitecturaalonafhankelijke modellering is dat er meerdere ontwerppunten kunnen geëvalueerd worden met eenzelfde statistisch profiel, wat de bruikbaarheid van statistische simulatie ten goede komt. In het geval van chip-multiprocessorsimulatie is het belangrijk om het fasegedrag van programma's nauwkeurig te vatten, aangezien dit een niet onbelangrijke invloed op totale prestatie heeft.

Deze bijdragen maken van statistische simulatie een snelle en nauwkeurige oplossing voor het simulatieprobleem van chip-multiprocessoren, en is in het bijzonder goed geschikt om de microarchitecturale ontwerpruimte te verkennen.

De gemiddelde fout op de prestatieschatting voor een chip-multiprocessor met één, twee, vier en acht kernen bedraagt respectievelijk 2.1%, 5.6% , 6.3% en 7.3%, terwijl de simulatie gemiddeld één tot vier grootteordes versneld wordt. Ondanks deze absolute fouten observeren we dezelfde prestatietrends met statistische simulatie als met cyclusgetrouwe simulatie. Bijvoorbeeld, statistische simulatie leidt tot dezelfde conclusie wanneer we de afweging maken tussen het aantal kernen enerzijds en de grootte van de gedeelde cache anderzijds. Bovendien identificeert het duidelijk welke benchmarks onderhevig zijn aan het delen van de cache tussen de verschillende processorkernen.

Intervalsimulatie is een nieuw, snel, nauwkeurig en eenvoudig te implementeren simulatieparadigma dat het abstractieniveau in architecturale simulatie verhoogt; het vervangt het cyclusgetrouwe model voor de processorkern door een mechanistisch analytisch model.

In het cyclusgetrouwe model worden de individuele instructies stap voor stap gevolgd doorheen de pijplijn van de superscalaire out-of-order processor, terwijl de intervalsimulator de eigenlijke stroom van instructies doorheen de pijplijn beschouwt. Intervalsimulatie pakt het simulatieprobleem op twee fronten aan: het doel is om de simulatiesnelheid te verbeteren én het ontwerp van een simulator te vereenvoudigen, zonder al te veel aan nauwkeurigheid in te boeten.

De inzichten verkregen uit intervalanalyse laten ons toe om nauwkeurig de timing van instructies te modelleren. De basis voor dit mechanistisch analytisch model is dat in een superscalaire out-of-order processor de instructies vlot doorheen de pijplijn stromen in afwezigheid van missers. Missers (cachemisser, foutief voorspelde sprong, etc.) delen de instructiestroom op in geïsoleerde intervallen; dit is het best waarneembaar in de dispatch-trap van de pijplijn. De analytische modellen voor de individuele kernen raadplegen de sprongvoorspellingen en de geheugenhiërarchiesimulatoren om de missers en de bijhorende latenties te bepalen – de lengte van een interval is afhankelijk van het type en de latentie van de misser. Zowel de prestatie op niveau van de processorkern als op niveau van het gehele systeem wordt bepaald door de lengte van deze intervallen, wat op zijn beurt de timing van toekomstige missers zal bepalen.

Onze experimentele resultaten tonen aan dat intervalsimulatie nauwkeurig is; we melden een gemiddelde schattingsfout van 4.6% voor de full-system simulatie van de meerdradige PARSEC benchmarks. Bovendien leidt intervalsimulatie tot correcte beslissingen in praktische studies, terwijl de simulatie een grootteorde sneller is dan de cyclusgetrouwe simulatie. Tevens is intervalsimulatie makkelijk te implementeren; ons model telt niet meer dan duizend regels code, in plaats van tienduizenden regels code in cyclusgetrouwe simulatoren. Aldus kunnen we besluiten dat intervalsimulatie een goede afweging maakt tussen ontwikkelingstijd, simulatietijd en nauwkeurigheid.

Summary

Designing a microprocessor is extremely time-consuming: it can take up to seven years before a next-generation processor hits the market. Computer architects heavily rely on cycle-level (and in many cases truly cycle-accurate) simulators in various stages of the design of a new processor, e.g., to drive high-level design decisions during early stage design space exploration. Architectural simulators model the microarchitecture in software, at some level of abstraction. The benefit of architectural simulators is that they yield relatively accurate performance results, are highly parameterizable and are very flexible to use. The downside, however, is that they are at least three or four orders of magnitude slower than real hardware execution.

While this is true for single-core superscalar out-of-order processor simulation, the current trend towards chip-multiprocessors or multicore processors, only exacerbates the problem. A multicore processor combines several cores on a single chip, sharing some resources such as last-level caches, off-chip bandwidth, main memory, etc. Co-executing threads affect each other's performance through the shared resources, e.g., conflict misses due to cache sharing may cause some threads to make slower progress than others. Moreover, changes in the microarchitecture may change which parts of the threads execute together, which in turn may lead to different conflict behavior and thus different relative progress rates for the co-executing threads. This tight performance entanglement between co-executing threads makes it hard to model performance of a multicore processor. As the number of cores on a multicore processor increases, simulation speed has become a major concern in computer architecture research and development.

Researchers and computer designers are well aware of the (multicore) simulation problem and have been proposing various methods for coping with it. In this dissertation, we propose and evaluate two simulation techniques, namely statistical simulation and interval simu-

lation, which both reduce the simulation time significantly.

Statistical simulation. The basic idea behind statistical simulation is to speedup the simulation by reducing the dynamic instruction count. Essentially, statistical simulation is performed in three steps. First, we measure a statistical profile of a program execution through functional simulation or through profiling; a statistical profile collects a number of program execution characteristics, such as instruction mix, inter-instruction dependence distributions, branch behavior information and memory behavior information. These statistics are then used to build a synthetic trace; this synthetic trace exhibits the same execution characteristics as the original program trace by construction, but it is much smaller in terms of its dynamic instruction count. Simulating this synthetic trace then yields a performance estimate. Given its short length (on the order of a couple millions of instructions), simulating a synthetic trace is done very quickly.

We make two contributions to the statistical simulation paradigm. First, we improve statistical simulation for single-core processors by accurately modeling the memory data flow in order to capture memory-level parallelism, secondary miss events, and load/store aliasing and bypassing. Second, we extend the statistical simulation paradigm to chip-multiprocessors running multiprogram workloads in order to capture the conflict behavior in shared resources. This requires that we model accesses to the memory hierarchy in a microarchitecture-independent way, using metrics as memory location reuse distance and stack distance. Moreover, microarchitecture-independent memory data flow modeling allows to evaluate more microarchitectural design points based on a single statistical profile, and thus improves the applicability of statistical simulation. Furthermore, we show that in case of multicore simulation it is important to accurately model time-varying program execution behavior, i.e., overall performance is affected by the phase behavior of the co-executing programs.

Both contributions make statistical simulation a fast and accurate solution to the multicore simulation problem, and make it a viable simulation approach to chip-multiprocessor design space exploration. For our baseline superscalar out-of-order architecture, we obtain an average overall performance estimation error of 2.1% for a single-core, 5.6% for a two-core, 6.3% for a four-core, and 7.3% for an eight-core processor, while achieving a simulation speedup of one order of magnitude. Despite these absolute errors, the same design space performance

trends are observed with statistical simulation as with cycle-accurate simulation. For example, statistical simulation leads to the same conclusion when making a trade-off between the number of cores on a chip and the size of the shared last-level cache. In addition, it clearly identifies which benchmarks are susceptible to cache sharing.

Interval simulation. Interval simulation is a novel, fast, accurate and easy-to-implement multicore simulation paradigm that raises the level of abstraction in architectural multicore simulation, i.e., it replaces the core-level cycle-accurate simulation model by a mechanistic analytical model. The core-level model of a truly cycle-accurate simulator tracks the individual instructions as they propagate through the pipeline. On the other hand, an interval simulator considers the stream of instructions through the dispatch stage of the pipeline. Interval simulation tackles the multicore simulation problem on two fronts: it reduces both the development and evaluation time, while not compromising accuracy too much.

Insights from interval analysis enable us to accurately model the timing of the instructions. The basis for this mechanistic analytical model is that a superscalar out-of-order core can smoothly stream instructions through its pipeline in the absence of miss events. Miss events however divide the smooth streaming of instructions into so called intervals; this is most clearly observed in the dispatch behavior of a program. The analytical timing models for the individual cores consult branch predictor, memory hierarchy and interconnection network simulators to derive miss events and their latencies. By analyzing the types of miss events and their latencies, we can derive the length of each interval, which determines core-level and system-level performance. The estimated core-level performance, in turn, drives the timing of (future) miss events.

Our experimental results show that interval simulation is fairly accurate; we report an average error of 4.6% for the multithreaded PARSEC benchmarks running in full-system simulation mode. Moreover, interval simulation leads to correct design decisions in practical design studies, while being one order of magnitude faster compared to cycle-accurate simulation. In addition, interval simulation is easy to implement; our model requires no more than one thousand lines of code, whereas a fully detailed simulator can easily consist of tens of thousand lines of code. Therefore, interval simulation makes a good balance between development time, simulation speed and accuracy.

Contents

List of Tables

List of Figures

List of Abbreviations

ALU	Arithmetic Logic Unit
ANTT	Average Normalized Turnaround Time
BTB	Branch Target Buffer
CMOS	Complementary Metal Oxide Semiconductor
CMP	Chip-Multiprocessor
CoV	Coefficient of Variation
CPI	Cycles per Instruction
CPU	Central Processing Unit
D-Cache	Data Cache
D-TLB	Data Translation Lookaside Buffer
DL1	Level-1 Data Cache
DL2	Level-2 Data Cache
DRAM	Dynamic Random Access Memory
ED^2P	Energy-Delay-Square Product
EPI	Energy per Instruction
FPGA	Field-Programmable Gate Array
I-Cache	Instruction Cache
I-TLB	Instruction Translation Lookaside Buffer
IL1	Level-1 Instruction Cache
IL2	Level-2 Instruction Cache
ILP	Instruction Level Parallelism
IPC	Instructions per Cycle
ISA	Instruction Set Architecture
KIPS	Kilo Instructions per Second
L1	Level-1 Cache
L2	Level-2 Cache
LSQ	Load Store Queue
MIPS	Million Instructions per Second
MLP	Memory-Level Parallelism
MSHR	Miss Status Holding Register

OS	Operating System
RAR	Read after Read
RAW	Read after Write
ROB	Reorder Buffer
SFG	Statistical Flow Graph
SMP	Shared-memory Multiprocessor
SMT	Simultaneous Multithreading
STP	System Throughput
TLB	Translation Lookaside Buffer
WAR	Write after Read
WAW	Write after Write

Chapter 1

Introduction

Anyone who has never made a mistake has never tried anything new.
Albert Einstein

Designing a microprocessor is extremely time-consuming: it can take up to seven years before a next-generation processor hits the market [56]. A long time-to-market is undesirable, not only from an economical point of view, but also from a technical perspective. Processor architects make most of the high-level design decisions during the early stages of the design process, using benchmarks and compilers that are available at the time. If too much time elapses before the final processor becomes available then it could very well be that the processor delivers suboptimal performance for emerging workloads that are different from the benchmarks that were used during the design process [80].

1.1 Microprocessor design challenges

In the following subsections, we will explain what causes this extremely long development time, and we will discuss why future technology trends worsen the problem. We will start with a brief overview of the history of a single-core processor, followed by a short introduction to chip-multiprocessors. We will then discuss the impact of processor technology and workload composition on architectural simulation and how it impacts development time.

1.1.1 Trends in single-core processor technology

According to Moore's law, transistor density on integrated CMOS circuits doubles about every two years. This pace has kept up for over forty years. Throughout the years, computer architects have introduced several techniques to improve processor performance by putting more and more components on the chip.

In its simplest form, the CPU executes one instruction at a time in a sequential way. However, the execution of an instruction can be broken into five more or less independent stages: (IF) fetch the instruction from memory, (ID) decode the instruction and generate the appropriate control signals, (EX) feed the control signals into the ALU and produce a result, (MEM) read from memory, (WB) write the result back to the register file or to memory. The instruction throughput can be improved by overlapping the execution of multiple instructions each in a different stage of their execution, e.g., while one instruction is being decoded, one can already fetch the next instruction from memory. This concept has led to pipelined architectures. The benefit is that the processor now consists of multiple stages with reduced complexity, allowing for higher clock rates. The classical pipeline has five stages, however, architectures with over ten pipeline stages are common for modern high-end processors.

The switch to pipelined architectures led to another concept, namely branch prediction and speculative execution. After a branch instruction is fetched, the CPU does not yet know which instruction address to fetch from in the next cycle, i.e., the direction and target of the branch are unknown before the branch gets to the writeback stage. However, we can try to predict what the next instruction address will be using a branch predictor. The CPU speculatively fetches instructions starting from the predicted address. If the target is predicted correctly then no time is wasted; if on the other hand, the target is mispredicted, the pipeline needs to be flushed and the execution restarts at the correct path.

In order to hide the access latency to main memory, processors now have a memory hierarchy. Multiple levels of on-chip and off-chip caches reside in between the execution core and main memory. Modern architectures typically have two or three levels of cache. The ones closer to the execution core tend to be smaller and thus faster than the ones further away. Caches are typically implemented as static RAM (or SRAM), which is much faster but more costly than the DRAM

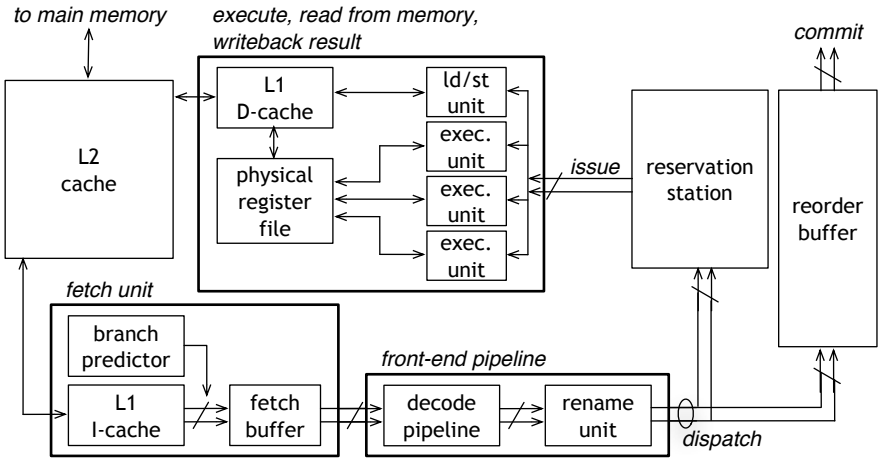

Figure 1.1: Diagram of a generic pipelined superscalar out-of-order processor.

technology used in main memory.

Superscalar out-of-order architectures boost performance by exploiting instruction level-parallelism (ILP). The processor can execute multiple independent instructions simultaneously and out of the original program order. Figure 1.1 shows a diagram of a typical contemporary high-performance out-of-order processor; in the next paragraph, we will give a short overview of the most important features.

The fetch unit fetches the instructions from the level-one (L1) instruction-cache (I-cache), and in case of a branch, it accesses the branch predictor to determine the next instruction address to fetch from. Subsequently, the instructions move into the fetch buffer before being decoded. The register rename unit removes all write-after-write (WAW) and write-after-read (WAR) dependences by dynamically mapping the instruction operands to real physical registers. Each stage can handle as many instructions per cycle as the bandwidth allows for, e.g., a fetch width of eight allows for eight instructions to be fetched per cycle. An instruction is dispatched into the reservation station (also called the issue queue) as long as the maximum dispatch width has not been exceeded, and provided that a reservation station entry is available. The reservation station entry keeps track of the read-after-write (RAW) dependences and issues the instruction to an available functional unit as soon as all input data dependences are resolved. It then becomes available for holding a newly dispatched instruction.

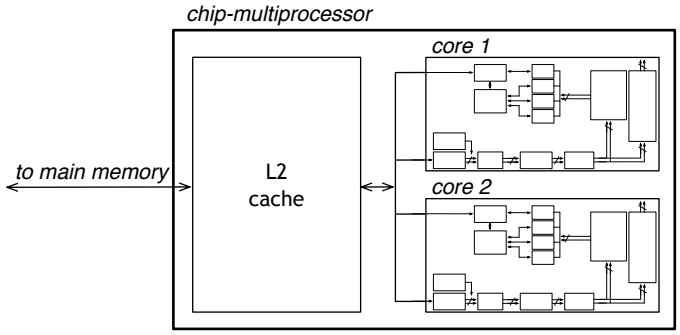

Figure 1.2: Diagram of a generic (two-core) chip-multiprocessor, sharing the L2 cache, off-chip bus, etc. between all cores.

The functional unit executes the instruction and writes the result back to the physical register file, or in case of a load/store unit, it accesses the L1 data-cache (D-cache). The reorder buffer (ROB) keeps track of the status of all in-flight instructions, i.e., instructions in the back-end pipeline which have not yet completed. The reservation station and the ROB can be two separate structures or can be merged into one so called instruction window. Either way, they allow for ILP to be exploited. In addition, memory-level parallelism (MLP) is exploited, i.e., multiple reads and/or writes access the main memory simultaneously. Non-blocking caches allow for multiple outstanding misses, hereto miss status holding registers (MSHRs) are used to administer each miss and subsequent delayed hits to the same cache line.

Superscalar out-of-order processors are available on the high-performance processor market for over a decade. Some examples are: DEC Alpha 21264, MIPS R10000, Intel P6 family (Pentium Pro, Pentium III, Core, Core2, Core i7), Intel Pentium 4, AMD K5 through K8 and K10, IBM Power 1 through 5.

1.1.2 The shift towards chip-multiprocessors

Many of the enhancements mentioned in the previous subsection relate to instruction-level parallelism being exploited in order to increase performance. Thread-level parallelism is another type of parallelism that can be exploited by recently introduced chip-multiprocessors (CMP); also called multicore processors, see Figure 1.2 for a diagram of a typical CMP with two cores. A multicore processor combines several cores

on a single chip, sharing some resources such as last-level caches, on-chip interconnection network, off-chip bandwidth and main memory. Today's microprocessors come with two to eight cores and it is to be expected that the number of cores will continue to increase. In fact, Intel has built a prototype with eighty cores during their (ongoing) 'tera-scale' computing research activities. Some examples of commercial CMPs are: Intel Core Duo, Core 2 Duo and Core i7, AMD K8 and K10, IBM Power 4 to 6 and Cell, Sun Niagara T1 and T2.

1.1.3 Architectural simulation

Computer architects use architectural simulation to drive design decisions early in the design cycle. At this stage it is infeasible and too costly to build hardware prototypes for performance evaluation studies. Architectural simulators model the microarchitecture in software, at some level of abstraction. The benefit of architectural simulators is that they yield relatively accurate performance results, are highly parameterizable and are very flexible to use.

The downside, however, is that they are at least three or four orders of magnitude slower than real hardware execution. Architectural simulators typically model the microprocessor in great detail, and very often, in a cycle-accurate manner. As more and more components of high complexity are modeled, more instructions need to be executed for each simulated instruction. As a result, culling a large design space through cycle-accurate simulation has become infeasible. While this is true for single-core processor simulation, the current trend towards chip-multiprocessors only exacerbates the problem. Several cores need to be simulated sequentially. In addition to single-core processor simulation, the simulator must also model the conflict misses and the contention for shared resources. Hence, simulation speed has become a major concern in computer architecture research and development as the number of cores on a multicore processor increases.

In addition, developing an architectural simulator at a high level of detail is tedious, costly and very time-consuming. Thus, one could or even should pose the question whether this level of detail is needed or called for during every stage in the design cycle.

1.1.4 Workload composition

When designing a new processor, one starts with defining the target domain: what kind of applications will typically be run on the processor? A representative workload, i.e., a set of benchmarks with their data input sets, is composed accordingly. The remainder of the design process is based on this workload. Past and recent trends in the software world have led to long-running benchmarks combined with large input sets [1, 35]—today's benchmarks execute several hundreds of billions or even trillions of instructions. This, in turn, has a negative impact on simulation time, i.e., more instructions need to be simulated, which prolongs the development time of new processors. Simulating an industry-standard benchmark to completion for a single microprocessor design point easily takes weeks to months, even on today's fastest machines and simulators.

1.2 Focus and contributions in this dissertation

In this dissertation we focus on the early stage design space exploration of contemporary high-performance processors. In this context, we propose and evaluate two simulation techniques, namely statistical simulation and interval simulation, which both reduce the simulation time significantly.

Before doing so, we would like to state that we do not envision these techniques as a replacement for cycle-accurate architectural simulation. Instead, we view both simulation paradigms as useful complements to the other tools a computer designer has at his/her disposal when designing a microprocessor. Moreover, statistical simulation as well as interval simulation are orthogonal to and can be used in conjunction with other existing simulation speedup approaches such as sampled simulation and FPGA-accelerated simulation.

1.2.1 Statistical simulation

Statistical simulation is a recently introduced approach for efficiently culling the microprocessor design space [19, 20, 23, 58, 59, 61]. The basic idea behind statistical simulation is to capture the essence of a real workload in a much shorter running synthetic trace. This is done in three steps. First, we measure a statistical profile of a program execu-

tion through (specialized) functional simulation or profiling; a statistical profile collects a number of program execution characteristics, such as instruction mix, inter-instruction dependence distributions, branch behavior information and memory behavior information. These statistics are then used to build a synthetic trace; this synthetic trace exhibits the same execution characteristics as the original program trace by construction, but is much smaller in terms of its dynamic instruction count. Simulating this synthetic trace then yields a performance estimate. Given its short length (on the order of a couple millions of instructions), simulating a synthetic trace is done very quickly.

1.2.2 Contribution #1: Accurate memory data flow modeling in statistical simulation

The memory subsystem has a significant impact on the overall performance of contemporary microprocessors. Poor modeling of the memory data flow may yield large performance prediction errors. The state-of-the-art in statistical simulation prior to this dissertation considers simple memory data flow modeling, showing three major shortcomings.

First, none of the prior work accurately models cache miss patterns, or the number of instructions in the dynamic instruction stream between misses. However, cache miss patterns have an important impact on the available memory-level parallelism. Independent long-latency misses that are close enough to each other in the dynamic instruction stream to make it into the reorder buffer together, (potentially) overlap their execution, thereby exposing memory-level parallelism. We accurately model cache miss patterns by collecting the cache characteristics dependent on a history of cache hit/miss outcomes.

Second, statistical simulation typically assigns hits and misses to loads and stores, and does not model delayed hits. A delayed hit, i.e., a hit to an outstanding cache line, is modeled as a cache hit although it should see the remaining latency of the outstanding cache line. We model delayed hits through cache line reuse distance distributions.

Third, none of the previously proposed statistical simulation approaches adequately model load bypassing and load forwarding, i.e., it is assumed that loads never alias with preceding stores. We model aliasing through read-after-write memory dependence distributions.

Accurately modeling the memory data flow reduces the average

prediction error in statistical simulation from 10.9% to 2.1%, while being 2 to 4 orders of magnitude faster than detailed cycle-accurate simulation. A discussion on the improved memory data flow modeling has been published in:

> Davy Genbrugge, Lieven Eeckhout and Koen De Bosschere, "Accurate Memory Data Flow Modeling in Statistical Simulation", *ICS '06: Proceedings of the 20th Annual International Conference on Supercomputing, pp.87-96, Jun. 2006*, [31].
>
> Davy Genbrugge and Lieven Eeckhout, "Memory Data Flow Modeling in Statistical Simulation for the Efficient Exploration of Microprocessor Design Spaces", *IEEE Transactions on Computers, Vol.57, No.1, pp.41-54, Jan. 2008*, [29].

1.2.3 Contribution #2: Multicore statistical simulation

Previous work has explored the statistical simulation paradigm extensively for single-core out-of-order processor simulation; and one earlier study [60] and one more recent study [38] applied statistical simulation to multithreaded workloads running on shared-memory multiprocessor systems. None of this prior work addresses the modeling of shared resources in chip-multiprocessors though. Co-executing threads affect each other's performance through inter-thread synchronization and communication, as well as through the shared resources. Resource sharing may cause some threads to run slower than others. Changes in the microarchitecture may change which parts of the threads execute together. This change, in turn, may lead to different conflict behavior in the shared resources, which may lead to different relative progress rates for the co-executing threads.

We extend the statistical simulation technique to chip-multiprocessors running multiprogram workloads. In order to capture the conflict behavior in shared resources, we model accesses to the memory hierarchy in a microarchitecture-independent way. Hereto, we use the notion of stack depth inspired by the least-recently-used (LRU) cache replacement policy. In addition, we show that it is important to accurately model time-varying program execution behavior, in order to accurately capture conflict behavior in shared resources.

Our results show that statistical simulation with the aforementioned enhancements is accurate and capable of tracking the trends in

a CMP design space. We report average performance prediction errors of less than 7.3%. A discussion on the modeling of conflict behavior in shared resources of a chip-multiprocessor has been published in:

> Davy Genbrugge and Lieven Eeckhout, "Statistical Simulation of Chip-Multiprocessors Running Multiprogram Workloads", *ICCD '07: Proceedings of the 25th International Conference on Computer Design, pp.464-471, Oct. 2007*, [28].
>
> Davy Genbrugge and Lieven Eeckhout, "Chip Multiprocessor Design Space Exploration through Statistical Simulation", *IEEE Transactions on Computers, Vol.58, No.12, pp.1668-1681, Dec. 2009*, [30].
>
> Lieven Eeckhout and Davy Genbrugge, Invited Chapter "Statistical Simulation", in "Processor, Multicore and System-on-Chip Simulation", *Olivier Temam and Rainer Leupers (Eds.), Springer, 2010*, [22].

1.2.4 Contribution #3: Interval simulation

Interval simulation is a novel, fast, accurate and easy-to-implement multicore simulation paradigm. Whereas other simulation techniques, including statistical simulation, increase simulation speed and have their place in the architect's toolbox, they model a multicore processor at a high level of detail which impacts development time and which may not be needed for many practical research and development studies. For example, when studying trade-offs in the memory hierarchy, cache coherence protocol or interconnection network of a multicore processor, cycle-accurate core-level simulation may not be needed. Interval simulation raises the level of abstraction in architectural multicore simulation to a level that makes multicore simulator development tractable and speeds up multicore simulation substantially, while not compromising accuracy too much.

Interval simulation replaces the core-level cycle-accurate simulation model in a multicore simulator by a mechanistic analytical model. The mechanistic analytical model drives the timing simulation of the individual cores without the detailed tracking of individual instructions through the cores' pipeline stages. The basis for the mechanistic analytical model is that a superscalar out-of-order core can smoothly stream instructions through its pipeline in the absence of miss events. Miss

events however divide the smooth streaming of instructions through the core's pipeline into so called intervals. The analytical timing models for the individual cores consult branch predictor, memory hierarchy and interconnection network simulators to derive miss events and their latencies. By analyzing the types of miss events and their latencies, we can derive the timing for each interval, which determines core-level and system-level performance.

The cooperation between the mechanistic analytical model and the miss event simulators enables the modeling of the tight performance entanglement between co-executing threads on multicore processors. We report average prediction errors below 6% while being one order of magnitude faster for multithreaded benchmarks running on a full-system simulator. Our core-level mechanistic analytical model is no more than one thousand lines of code versus twenty eight thousand lines for our detailed core-level model. An extensive discussion of this new simulation paradigm is presented in:

> Davy Genbrugge, Stijn Eyerman and Lieven Eeckhout, "Interval Simulation: Raising the Level of Abstraction in Architectural Simulation", Accepted for publication in *HPCA '10: Proceedings of the 16th International IEEE Symposium on High-Performance Computer Architecture, Jan. 2010*, [32].

1.3 Thesis outline

This dissertation is organized as follows. In Chapter 2 we discuss architectural simulation and give an overview of other existing fast simulation techniques. Chapter 3 describes the basics of the statistical simulation paradigm as we envision it, followed by a detailed discussion in Chapter 4 on how to accurately model memory data flow (Contribution #1). Chapter 5 discusses the enhancements to the statistical simulation paradigm for the purpose of CMP design space exploration running multiprogram workloads (Contribution #2). In Chapter 6 we discuss interval simulation and we show how raising the level of abstraction in architectural simulation addresses the two problems of simulator development time and simulation speed (Contribution #3). Finally, we conclude in Chapter 7 and we give some suggestions for future work.

Chapter 2

Architectural Simulation

A good simulation, be it a religious myth or scientific theory, gives us a sense of mastery over experience. To represent something symbolically, as we do when we speak or write, is somehow to capture it, thus making it one's own. But with this appropriation comes the realization that we have denied the immediacy of reality and that in creating a substitute we have but spun another thread in the web of our grand illusion.
Heinz Rudolf Pagels

Computer architects in industry and academia heavily rely on cycle-level (and in some cases truly cycle-accurate) simulators, i.e., architectural simulators are used at various stages during the design of a new processor. However, architectural cycle-level simulation is very costly in terms of simulation time as well as development time at the earliest stages of the design.

Architectural simulators model the microarchitecture at some level of abstraction, i.e., they provide detailed software models of processor features such as caches, branch predictors, instruction windows, reorder buffers, load/store queues, etc. Typically, individual instructions are tracked as they propagate through the processor's pipeline. Industrial single-core simulators typically simulate one thousand to ten thousand (target) cycles per second (1 KHz to 10 KHz); academic simulators typically run ten to a few hundred kilo instructions per second (KIPS) [14]. The input for these simulators are entire benchmarks, executing hundreds of billions of instructions. Multicore processor simulators exacerbate the problem because they have to simulate multiple cores, and have to model inter-core communication (e.g., coherence

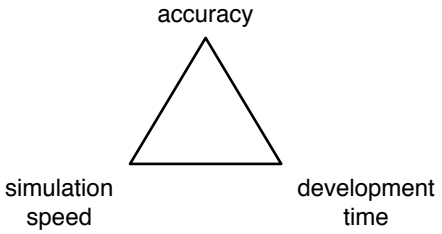

Figure 2.1: Trade-offs in architectural simulation and modeling.

traffic) as well as resource contention in shared resources such as shared caches, interconnection network, chip I/O, etc.

Researchers and computer designers are well aware of the (multi-core) simulation problem and have proposed various methods for coping with it. Figure 2.1 illustrates the trade-offs between accuracy, simulation speed and simulator development time one has to make depending on the objective of a simulation technique. For example, when making system-level design decisions early in the design process, too much detail only gets in the way. One can then choose to trade accuracy for simulation speed and development time, in order to define the high-level microarchitecture more quickly. On the other hand, when studying a particular low-level microarchitectural feature in detail, e.g., memory dependence prediction, one needs a more accurate but slower simulator, which is more costly to build.

Computer architects have a set of evaluation tools in their toolbox, often used in combination, i.e., many of the techniques are orthogonal. These tools include: functional simulation and full-system simulation, specialized cache and branch predictor simulation, trace-driven and execution-driven detailed cycle-accurate simulation, and many techniques to speedup the simulation such as sampling, parallelization, hardware acceleration, statistical and analytical modeling.

2.1 Functional simulation and full-system simulation

Basically, a functional simulator implements an instruction set architecture (ISA), i.e., it emulates how the output operand of an instruction is computed from its input operands. Functional simulation is most

useful for determining whether the design implements an instruction set correctly and whether a software program, including the operating system (OS), behaves as expected. It mimics the behavior of a microprocessor or the entire computer system. In other words, for a given program and its input, the emulator produces the same result as a real hardware execution. Functional simulation does not model any timing aspects, nor does it model the microarchitectural components. Consequently, functional simulation has poor accuracy with respect to performance evaluation, but on the other hand it is very fast. Moreover, the development cost is small because functional simulators have a long lifetime; a new implementation is needed only when the instruction set architecture changes.

For many programs, including the SPEC CPU benchmark suite, it is sufficient to emulate the system calls, without actually executing the kernel-space code. However, some applications such as webservers and databases require that the complete software stack from a real system runs on the simulator without modification. Moreover, the recent shift towards multicore processors running multithreaded applications drives the need for the simulation of the entire computer system, including the OS, device drivers, etc. (In particular, OS thread scheduling can affect program behavior.) Basically, a full-system simulator implements an instruction set emulator and a virtual hardware layer capable of booting an operating system. Examples of simulators with full-system support are SimOS [66], Simics [53], M5 [6] and PTLSim [81].

2.2 Specialized cache and branch predictor simulation

When studying caches or branch predictors in isolation, one can use a specialized simulator. A functional simulator generates an instruction or address stream which is fed into the specialized simulator. The specialized simulator models the cache hierarchy or the branch predictor in detail, but no other processor features are implemented.

Specialized simulation yields accurate cache miss rates or branch misprediction rates. However, overall performance accuracy is poor due to the lack of modeling other microarchitectural features which may have impact on the performance as well. The main benefit of these types of simulators lies in the low development cost and the high sim-

ulation speed. SimpleScalar [9] and M5 [6] are two example simulation tool sets which provide specialized simulation models. Sugumar and Abraham [70] show how multiple caches under optimal replacement can be simulated simultaneously. Other well-known cache simulators are DineroIV [18] and cachesim [10].

2.3 Trace-driven and execution-driven cycle-accurate simulation

Highly accurate simulation models most of the microarchitectural features in a cycle-accurate manner. Cycle-accurate simulation comes in two variants: trace-driven simulation and execution-driven simulation.

Trace-driven simulation separates the functional simulation from the timing simulation. A benchmark trace is generated with a functional simulator. This trace is fed into the detailed architectural timing simulator. The benefit of this approach is that a benchmark needs to be functionally simulated only once. A drawback is that these traces can be very large, requiring a lot of disk space. In addition, the traces do not incorporate instructions along mispredicted paths. Hence, it is impossible to model the effects these off-path instructions may have on performance.

Execution-driven simulation combines functional simulation and detailed architectural timing simulation, e.g., SimpleScalar [9], M5 [6], PTLSim [81], etc. The simulator takes the original binary program as input and executes it as it would be done on the real hardware. In such way, off-path instructions are simulated as well and the timing simulation captures the possible interference off-path instructions have on the overall performance.

Cycle-accurate simulation is very slow because it simulates all the processor features in detail. For the same reason it requires a long time to develop. Fortunately, the development time can be improved by providing a modular simulation infrastructure, such as Asim [24], Liberty [72] and Unisim [2].

2.4 Accelerating simulation

As mentioned before, architectural simulation is very time-consuming. Throughout the years, researchers have proposed various techniques to speed up simulation. We will discuss some of these techniques in the following subsections.

2.4.1 Sampled simulation

The idea of sampled simulation is to simulate a limited number of sampling units instead of the entire dynamic instruction stream. A sampling unit is a (small) fragment of consecutive instructions that is run on a detailed cycle-accurate simulator. One can use functional simulation in order to fast-forward between sampling units or use architectural checkpoints per sampling unit.

The major difficulty of sampled simulation is to select representative sampling units, either one big sample or multiple small samples. A more general problem is that the execution of a program consists of several program phases. Therefore, sampling units must be chosen such that they represent each major program phase. The sampling units are selected either randomly (Conte et al. [17]), or periodically (SMARTS by Wunderlich et al. [79]), or based on phase analysis (SimPoint by Sherwood et al. [67]).

Sampled simulation is fast and yields accurate performance estimates. Recent advances in architecture and microarchitecture state (re-)construction prior to each sampling unit enable the simulation of single-threaded benchmarks in the order of minutes with an error of around a few percent [73, 78].

2.4.2 Statistical simulation

The main objective of statistical simulation is similar to sampled simulation, i.e., the goal is to reduce the number of instructions that need to be simulated. Statistical simulation first records the distributions of important program characteristics into a statistical profile. The statistical profile serves as input for a synthetic trace generator, which generates a much smaller trace compared to the original full program trace. This synthetic trace statistically resembles the original trace by construction. Simulating this synthetic trace on a cycle-accurate simulator yields per-

formance estimates.

Although statistical simulation is (slightly) less accurate compared to cycle-accurate simulation of the full program trace, it is still capable of accurately tracking performance trends throughout the design space. Moreover, simulating a synthetic trace is done very quickly because it is many orders of magnitude smaller than the full program trace.

Chapter 3 elaborates on the statistical simulation paradigm, which forms the basis for the work presented in Chapters 4 and 5.

2.4.3 Analytical modeling

There are basically three approaches to analytical performance modeling: empirical modeling, mechanistic modeling and hybrid empirical/mechanistic modeling.

Empirical modeling, also called black-box modeling, learns a performance model through training and does not assume specific knowledge about the target processor. Ipek et al. [39] learn a model through neural networks, and Lee and Brooks [50] build a model through regression modeling. Lee et al. [51] leverage regression modeling to predict multiprocessor performance running multiprogram workloads.

Mechanistic modeling [26, 46, 47, 54, 69, 71] constructs a model by looking into the mechanisms in the target processor that affect the performance. Michaud et al. [54], Karkhanis and Smith [46], Taha and Wills [71] estimate performance through interval-based models which focus on the processor's issue rate. By modeling the performance in terms of the processor's dispatch behavior, Eyerman et al. [26] significantly simplify these interval models. These first-order core-level performance models serve as the basis for interval simulation which we describe in Chapter 6.

Hybrid mechanistic/empirical modeling proposes a mechanistic performance formula in which the parameters are derived through empirical modeling, see for example the pipeline model by Hartstein and Puzak [34].

2.4.4 Parallelized and/or hardware-accelerated simulation

Architectural simulation is intrinsically highly parallelizable because it models hardware that has much inherent parallelism. By exploiting the coarse-grained parallelism in the simulator, one can signifi-

cantly speedup the simulation. For example, the parallel Wisconsin Wind Tunnel II [57] achieves a speedup of factor five when simulating a thirty-two core target machine on an eight-core host processor versus a single-core host processor. A disadvantage, is that a parallelized architectural simulator is tedious to develop. Penry et al. [63] build a structural model of a CMP that enables them to automatically parallelize the simulator. The individual components in the structural CMP model are designed to execute concurrently in hardware and are thus candidates to run in parallel in simulation.

In the recent years researchers have looked at field-programmable gate-arrays (FPGA) to speed up simulation, see for example RAMP by Wawrzynek et al. [77], FAST by Chiou et al. [15], Pellauer et al. [62] and Penry et al. [63]. FPGA-accelerated simulation speeds up simulation by mapping cycle-accurate timing models onto FPGAs. The simulation speedup comes from exploiting fine-grain parallelism in the FPGA. FPGA-integrated simulators typically run at a speed of a few million instructions per second (MIPS).

2.4.5 Interval simulation

As mentioned before, we propose a new simulation paradigm in this dissertation, namely, interval simulation. Interval simulation bridges the gap between detailed cycle-accurate simulation and mechanistic analytical performance modeling; it combines an interval-based core-level model [26] with specialized memory hierarchy, inter-connection network and branch predictor simulation. In other words, the analytical core-level model replaces the detailed core-level model in a truly cycle-accurate simulator. In such way, it raises the level of abstraction.

Unlike all other techniques mentioned in this section, interval simulation tackles the CMP simulation problem on two fronts, i.e., besides achieving a substantial simulation speedup, it significantly simplifies the simulator reducing development time and cost. In addition, interval simulation does not compromise accuracy too much, i.e., it yields accurate performance estimates, and allows to make accurate high-level microarchitecture design decisions.

Chapter 6 describes interval simulation in more detail.

Technique	Development time	Evaluation time	Accuracy	Multicore
Functional simulation	excellent	good	poor	yes
Specialized simulation (cache & branch predictor)	very good	good	poor	yes
Cycle-accurate simulation	poor	poor	excellent	yes
Sampled simulation	poor	good	good	yes
Analytical modeling	excellent	excellent	good	no
Parallelized and hardware-accelerated simulation	very poor	good	excellent	yes
Statistical simulation	poor to good	very good	good	yes
Interval simulation	very good	good	good	yes

Table 2.1: Comparing simulation and modeling techniques in terms of development time, evaluation time, accuracy, and ability to model multicore processors.

2.5 Overview and Discussion

Table 2.1 gives an overview of the simulation and modeling techniques in the toolbox of a computer architect. It summarizes the trade-offs of different techniques in terms of development time, evaluation time, and accuracy, and the ability to model multicore processors. It clearly illustrates the role of statistical simulation and interval simulation. Both techniques cope with the simulation speed problem, however, they tackle the problem on different sides: statistical simulation reduces the number of instructions that need to be simulated, whereas interval simulation reduces the number of instructions that need to be executed on the host in order to simulate one instruction. In addition, interval simulation also tackles the development time problem.

Chapter 3

Statistical Simulation: State-of-the-Art

Statistics are like bikinis.
What they reveal is suggestive, but what they conceal is vital.
Aaron Levenstein

Statistical performance modeling has gained a lot of interest over the past few years. This chapter describes the state-of-the-art in statistical simulation prior to this dissertation [19].

3.1 Single-core statistical simulation

Statistical simulation consists of three steps as shown in Figure 3.1. We first compute a *statistical profile* (i) through specialized functional simulation and/or through profiling, using (dynamic) binary instrumentation. This profile contains a number of important program execution characteristics such as control flow behavior, instruction mix, inter-instruction dependences, branch miss behavior, cache miss behavior, and TLB miss behavior. Subsequently, we use this statistical profile to generate a *synthetic trace* (ii) that is much smaller than the original program trace from which the profile was generated. In Chapter 4, we show that the synthetic trace can be up to 4 orders of magnitude smaller than the original program trace. The synthetic trace exhibits the same execution characteristics as the original program trace by construction. In the final step we simulate this synthetic trace (iii) on a statistical sim-

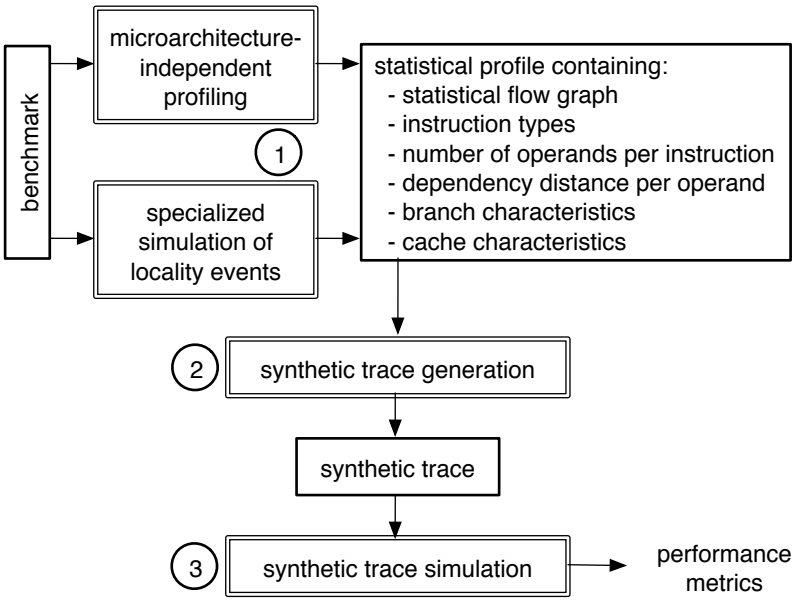

Figure 3.1: Statistical simulation: general framework.

ulator yielding performance metrics such as IPC. Given its short length, the synthetic trace rapidly runs to completion.

3.1.1 Statistical profiling

In statistical profiling we make a distinction between microarchitecture-*in*dependent characteristics and microarchitecture-dependent characteristics. The microarchitecture-independent characteristics can be used across microarchitectures during design space exploration. The microarchitecture-dependent characteristics on the other hand are particular to a specific microarchitecture component configuration. Ideally, the profile should contain only microarchitecture-independent characteristics, such that it applies to many microarchitectures and thus needs to be determined only once. Figure 3.2 illustrates what a statistical profile looks like; we now discuss each component in more detail.

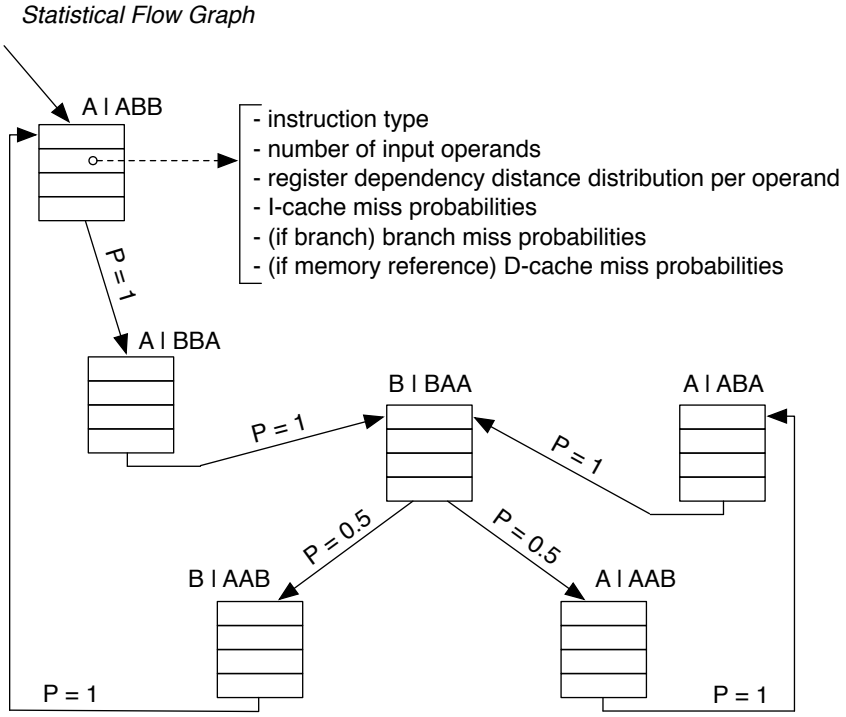

Figure 3.2: Illustration of the statistical profile. The notation 'A|ABB' represents basic block A with its history of three preceding basic blocks ABB.

Microarchitecture-independent characteristics

The key structure in the statistical profile is the *statistical flow graph (SFG)* [19] which represents a program's control flow behavior in a statistical manner. In an SFG, the nodes are the basic blocks along with their basic block history, i.e., the basic blocks being executed prior to the given basic block. The order of the SFG is defined as the length of the basic block history, i.e., the number of predecessors to a basic block in each node of the SFG. The order of an SFG will be denoted with the symbol k throughout this dissertation—throughout this chapter we consider third-order SFGs unless stated otherwise. For example, consider the following basic block sequence 'ABBAABAABBA'. The third-order SFG then makes a distinction between basic block 'A' given its basic block history 'ABB', 'BBA', 'AAB', 'ABA'; the SFG will thus contain the following nodes: 'A|ABB', 'A|BBA', 'A|AAB' and 'A|ABA'. The edges in the SFG interconnecting the nodes represent transition proba-

bilities between the nodes. Figure 3.2 gives an example third-order SFG for the aforementioned basic block sequence.

The idea behind the SFG is to model all the other program characteristics along the nodes of the SFG. This allows for modeling program characteristics that are correlated with (or dependent on) execution path behavior. This means that for a given basic block, different statistics are computed for different basic block histories, i.e., we collect different statistics for basic block 'A' given its history 'AAB' and 'ABB'. For example, the probability for a cache miss for a given load in basic block 'A' might be different depending on its basic block history. On the other hand, in case the correlation between program characteristics spans a number of basic blocks that is larger than the SFG's order, it will be impossible to model such correlations within the SFG, unless the order of the SFG is increased. However, in the next chapter we will show that it is possible to decouple cache miss correlation from the order of the SFG, i.e., we capture correlation that spans a number of basic blocks larger than the SFG's order.

The second microarchitecture-independent characteristic is the *instruction mix*. We classify the instruction types into 16 classes according to their semantics: nop, trap, load, store, software prefetch, write hint, integer conditional branch, floating-point conditional branch, indirect branch, integer arithmetic and logical operation, integer multiply, integer divide, floating-point arithmetic and logical operation, floating-point multiply, floating-point divide and floating-point square root. This distinction is made based on the instruction's semantics and its execution latencies. For each instruction we also record the number of *input registers* or source operands. Note that some instruction types, although classified within the same instruction class, may have a different number of source operands.

For each operand we also record the *dependence distance* which is the number of dynamically executed instructions between the production of a register value (register write) and its consumption (register read). We only consider read-after-write (RAW) dependences since our focus is on out-of-order architectures in which write-after-write (WAW) and write-after-read (WAR) dependences are dynamically removed through register renaming as long as enough physical registers are available. Note that recording the dependence distance requires storing a distribution since multiple dynamic versions of the same static instruction could result in multiple dependence distances. Although

very large dependence distances can occur in real program traces, for our purposes we can limit the dependence distances in the distribution to the maximum reorder buffer size we want to consider during statistical simulation. In our study, we limit the dependence distance to 512 which allows for modeling a wide range of microprocessors.

Microarchitecture-dependent characteristics

In addition to the microarchitecture-independent characteristics mentioned above, we also measure a number of microarchitecture-dependent characteristics that are related to locality events. The reason for choosing to model these events in a microarchitecture-dependent way is that locality events are hard to model using microarchitecture-independent metrics. We therefore take a pragmatic approach and collect cache miss and branch miss information for particular cache configurations and branch predictors.

For the *branch statistics* we consider (i) the probability of a taken branch, (ii) the probability of a fetch redirection (target misprediction in conjunction with a correct taken/not-taken prediction for conditional branches), and (iii) the probability of a branch misprediction. When measuring the branch statistics we consider a FIFO buffer as described in [19] in order to model delayed branch predictor update.

The *cache statistics* consist of the following six probabilities: (i) the L1 I-cache miss rate, (ii) the L2 cache miss rate due to instructions only[1], (iii) the L1 D-cache miss rate, (iv) the L2 cache miss rate due to data accesses only, (v) the I-TLB miss rate and (vi) the D-TLB miss rate.

3.1.2 Synthetic trace generation

The second step in the statistical simulation technique is to generate a synthetic trace from the statistical profile. The synthetic trace generator takes as input the statistical profile and outputs a synthetic trace that is fed into the statistical simulator. Synthetic trace generation uses random number generation for generating a number in the interval $[0, 1]$; this random number is then used with the inverse cumulative distribution function to determine the particular value for the program characteristic, see Figure 3.3.

[1] We assume a unified last-level L2 cache. However, we make a distinction between L2 cache misses due to instructions and due to data.

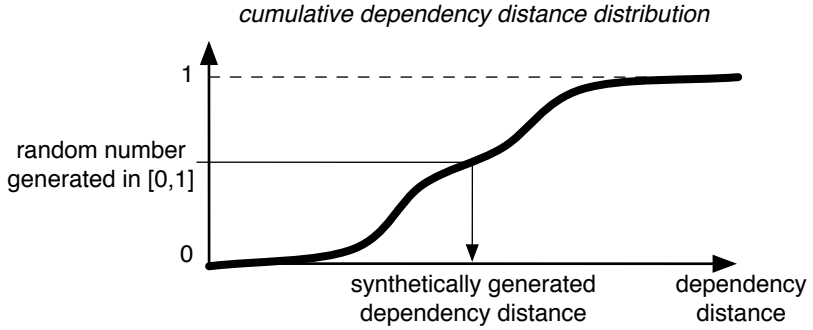

Figure 3.3: Illustrating synthetic trace generation using random number generation and the cumulative dependence distance distribution.

In particular, synthetic trace generation walks the SFG in a statistical way, i.e., for each node in the SFG, it determines the next node based on the inter-node transition probabilities. For each node, it outputs the instructions. The synthetic trace is the resulting linear sequence of the synthetic instructions. Synthetic trace generation determines the dependence distance for each input register for each instruction. In other words, it determines on which prior instruction this instruction depends through a RAW register dependence. Furthermore it labels the I-cache miss information, i.e., it describes whether the instruction fetch results in an L1 hit, L2 hit or L2 miss and whether the memory access generates a TLB miss. In case of a branch, the generator labels it as taken or not taken. Additionally, the generator determines whether it is a fetch redirected, a mispredicted or a correctly predicted branch. Loads and stores are assigned with D-cache miss information similar to the I-cache miss labels.

3.1.3 Synthetic trace simulation

Simulating the synthetic trace is fairly straightforward. In fact, the synthetic trace simulator itself is very simple as it does not need to model branch predictors nor cache hierarchies; also, all the ISA's instruction types are collapsed in a limited number of instruction types.

Instruction scheduling and execution is done in a way similar to conventional cycle-accurate architectural simulation. Instructions are scheduled for execution on a functional unit when their dependences have been resolved, and they are steered towards a specific functional

unit based on their instruction type. The instruction gets assigned a latency corresponding to the functional unit's operation latency, except for load instructions, for which the latency depends on their D-cache label and D-TLB label.

A load instruction's memory access latency is the maximum of the D-cache latency and the D-TLB latency. If the D-cache label is an L1 hit, an L2 hit or an L2 miss, then the D-cache latency equals the L1 access latency, the L2 access latency or the main memory access latency, respectively. If the load instruction incurs a D-TLB miss then the D-TLB latency equals the D-TLB miss latency. In the case of an I-cache miss, the fetch engine stops fetching for a number of cycles equal to the miss penalty.

Branches are labeled in the synthetic trace. The label states whether the branch is taken, fetch redirected or mispredicted. It determines the action the statistical simulator should take, similar to conventional architectural simulation. In particular, depending on the aggressiveness of the instruction cache fetch policy, fetch may stop upon a taken branch, or fetch may be redirected. On a branch misprediction, synthetic instructions are fed into the pipeline as if they were from the correct path. When the branch is resolved, the pipeline is flushed and refilled with synthetic instructions from the correct path. This is to model resource contention in case of a branch misprediction. Note that the statistical profile mentioned above does not consider off-path instructions; the statistics only concern on-path instructions.

The important benefit of statistical simulation is that the synthetic traces are very short. The performance metrics such as IPC quickly converge to a steady-state value when simulating a synthetic trace. Synthetic traces containing a few million instructions are sufficient for obtaining both stable and accurate performance estimations.

3.2 Discussion on applicability

The use of microarchitecture-dependent characteristics in the statistical profile limits the applicability of statistical simulation for design space exploration. For example, whenever a new branch predictor or a new cache hierarchy is to be considered, we need to collect a new statistical profile. To address this issue in the context of caches, techniques can be used for measuring cache profiles for multiple caches simultaneously in a single profiling run [36, 70]. However, a better solution to

recompute profile	*microarchitecture structure and/or configuration*
yes	cache hierarchy (number of cache levels, size, associativity, line size, replacement policy, line updating policy)
	branch predictor (type and size)
no	processor width (fetch, decode, dispatch, issue and commit width)
	pipeline depth (front-end pipeline depth)
	ROB size
	LSQ size
	fetch buffer size
	number and types of the functional units
	instruction execution latencies
	memory hierarchy (L1, L2 and DRAM) access latencies

Table 3.1: Example microarchitectural parameters that do or do not require that a new statistical profile is computed.

improve the applicability of statistical simulation would be to replace the microarchitecture-dependent characteristics by microarchitecture-independent characteristics, because the same profile could then be used across the entire design space, i.e., the profile is independent of the microarchitectural configuration. In this dissertation we will replace the microarchitecture-dependent cache statistics by (almost) microarchitecture-independent cache statistics, see Chapter 5 on statistical simulation for CMP design space explorations.

Since a statistical profile already contains a number of microarchitecture-independent characteristics, a very large number of microarchitectural parameters can still be varied during design space exploration without having to recompute the statistical profile, for example pipeline depth, processor width, and ROB/LSQ size, see Table 3.1. This allows statistical simulation to yield substantial speedups during design space exploration.

A second note that we would like to make is that this approach is orthogonal to sampled simulation approaches such as SimPoint [64, 67]. SimPoint, as an example sampling approach, selects a number of repre-

sentative simulation points over the entire program execution. Statistical simulation can then be applied to the individual simulation points. The important advantage of statistical simulation is that the number of simulated instructions is smaller than for SimPoint. In addition, statistical simulation simulates the branch predictor and the cache hierarchy in a statistical manner which is faster than the detailed simulation as required by SimPoint. However, statistical simulation needs to recompute the statistical profile when the cache or branch predictor is changed during design space exploration; this is not the case for SimPoint. Nevertheless, collecting a statistical profile is faster than running a detailed processor simulation. So, in summary, statistical simulation is faster than SimPoint and both techniques are orthogonal.

3.3 Other work in statistical modeling

Noonburg and Shen [58] have proposed to model program execution as a Markov chain in which the states are determined by the microarchitecture and the transition probabilities by the program. They have done so for a simple superscalar processor. However, extending their approach to large-resource out-of-order architectures is infeasible because of the exploding complexity of the Markov chain.

Iyengar et al. [40, 41] have taken a different approach in SMART. They use a statistical control flow graph to identify representative trace fragments; these trace fragments are extracted from the real program trace and are coalesced to form a reduced program trace. The statistical control flow graph uses the notion of a fully qualified instruction. A fully qualified instruction is an instruction along with its context. The context of a fully qualified instruction consists of its n preceding singly qualified instructions. A singly qualified instruction is an instruction along with its instruction type, I-cache behavior, TLB behavior, and if applicable, its branching behavior and D-cache behavior. SMART makes a distinction between two fully qualified instructions that have the same history of preceding instructions, however, they differ in a singly qualified instruction; that singly qualified instruction can be a cache miss in one case while being a hit in another case. Modeling a program execution using fully qualified instructions requires a lot of memory space to collect the statistical profile: the authors have reported that for some benchmarks, information needed to be erased from the statistical profile in order not to exceed the amount of mem-

ory available in the machine they have done their experiments on.

More recently, a number of papers have been published on the statistical simulation framework as we envision it in this dissertation. The idea is to collect a number of program characteristics and to generate a synthetic trace from them. This trace is then simulated on a simple trace-driven statistical simulator. The initial models proposed along this approach [11, 20, 21] are fairly simple in the sense that mostly aggregate statistics are used to model the program execution; these approaches do not model characteristics at the basic block level. Oskin et al. [61] have proposed the notion of a graph with transition probabilities between the basic blocks while still using aggregate statistics. Follow-on work has introduced a more fine-grained program characterization. Nussbaum and Smith [59] have correlated various program characteristics to the basic block size in order to improve accuracy. Eeckhout et al. [19] have proposed the statistical flow graph (SFG) which models a program's control flow in a statistical manner and which captures path-dependent program characteristics, i.e., correlated to the SFG.

A few papers have extended the single-processor statistical simulation paradigm to multithreaded programs. Nussbaum and Smith [60] have adapted statistical simulation for shared-memory multiprocessor (SMP) systems. To do so, they have extended statistical simulation to model synchronization and accesses to shared memory. Hughes and Li [38] more recently have introduced synchronized statistical flow graphs that incorporate inter-thread synchronization. Cache behavior is still modeled based on cache miss rates though; consequently, these proposals are unable to model shared caches as observed in modern CMPs.

Petoumenos et al. [65] have proposed StatShare, a statistical model for shared caches, which is derived from StatCache [4]. StatCache uses sparse data samples collected during a single run to estimate the cache miss rate assuming a fully-associative cache with random replacement. Chandra et al. [12] have proposed performance models to predict the impact of cache sharing on co-scheduled programs. The output provided by the performance model is an estimate of the number of extra cache misses for each thread due to cache sharing. These performance models are limited to predicting cache sharing effects, and they do not predict overall performance. Moreover, the performance models assume that co-scheduled programs make fixed progress, i.e., the models

ignore the effect cache sharing may have on how programs affect each other's performance.

Recent work also has focused on generating synthetic benchmarks rather than synthetic traces. Hsieh and Pedram [37] have generated a fully functional program from a statistical profile. However, all the characteristics in the statistical profile are microarchitecture-dependent, which makes this technique useless for microprocessor design space explorations. Bell and John [3] have generated short synthetic benchmarks using a collection of microarchitecture-independent and microarchitecture-dependent characteristics similar to what is done in statistical simulation. This work is further improved by Joshi et al. [44]: they use only microarchitecture-independent workload characteristics, allowing them to use the synthetic benchmark across a wide range of microarchitectures. Their goal is performance model validation using small but representative synthetic benchmarks, which cannot be reverse-engineered.

Chapter 4

Accurate Memory Data Flow Modeling in Statistical Simulation

The obstacle is the path.
Zen Buddhist Proverb

As mentioned in the introduction, previously proposed statistical simulation approaches assume relatively simple memory data flow statistics. In this chapter we propose three additional memory data flow modeling features: (i) cache miss correlation, (ii) through-memory read-after-write dependence distributions, and (iii) cache line reuse distributions.

4.1 Three shortcomings

The performance of a processor is determined to a great extent by the performance of the memory subsystem. Therefore, poor modeling of the memory data flow results in an inaccurate performance model for the entire processor system. The state-of-the-art in statistical simulation as described in Chapter 3 considers simple memory data flow modeling, which has three major shortcomings.

First, statistical simulation does not capture the possible correlation between cache misses. Cache misses often occur in certain patterns that have an important impact on the available memory-level parallelism.

Independent long-latency misses that are close enough to each other in the dynamic instruction stream to make it into the reorder buffer together, overlap their execution, thereby exposing memory-level parallelism (MLP).

Second, statistical simulation assumes that load addresses never alias with preceding store addresses. Under this assumption, loads can always bypass any store earlier in the load/store queue, which is not the case in real systems. More accurate modeling would be that a load does not bypass an aliasing store. However, it may retrieve its data directly from the store in the load/store queue without having to access the memory subsystem (load forwarding).

Third, loads and stores are profiled through a specialized cache simulator which does not take timing into account. Hence, a load is either a hit or a miss, and in contrast to real systems delayed hits do not occur. A delayed hit, i.e., a hit to an outstanding cache line, is modeled as a cache hit although it should see the latency of the outstanding cache line.

In the following sections we will discuss how we overcome these shortcomings, which leads to more accurate memory data flow modeling in statistical simulation.

4.2 Cache miss correlation modeling

Cache miss correlation refers to the fact that the cache miss behavior of a particular memory operation (load or store) is highly correlated with the cache miss behavior of (a) preceding memory operation(s). Consider for example a loop that walks over an array. Each element in the array is 8 bytes long and a cache line is 32 bytes long. As a result, a cache miss will occur every four iterations of the loop assuming the array is not residing in the cache. This is not accurately modeled in existing statistical simulation frameworks. The cache statistics for all iterations of this loop will collapse in a single number, namely the cache miss rate, which is 25%. During synthetic trace generation, this single cache miss rate number will result in a cache miss every four loop iterations *on average*—there is a variable number of cache hits between two misses because of the use of random numbers during synthetic trace generation. Through cache miss correlation modeling on the other hand, the synthetic trace will show one cache miss followed by exactly three cache hits, i.e., the number of cache hits between two

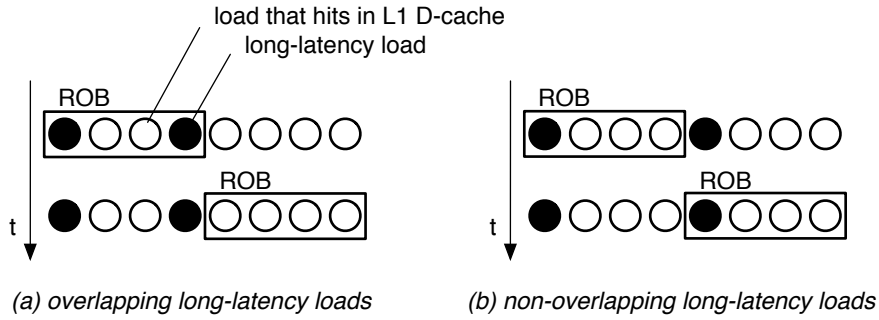

Figure 4.1: Illustrating the importance of the accurate modeling of overlapping long-latency loads. A trace of load instructions is shown as well as a window (ROB) sliding over it; a ROB size of four is assumed in this example.

misses is constant.

Accurately modeling the distance between misses is important because it has an immediate impact on the amount of memory-level parallelism (MLP) that can be exploited, which is illustrated in Figure 4.1. In both cases, see Figure 4.1 (a) and (b), there are 2 independent long-latency cache misses out of the 8 loads, i.e., the cache miss rate equals 25% in both cases. In case (a), the long-latency loads are closer to each other in the dynamic instruction stream than the number of entries in the ROB, and both long-latency loads will overlap in time provided that enough miss status holding registers (MSHRs) are available, i.e., memory-level parallelism is exposed [16, 45]. However, in case (b), the penalties for both long-latency loads will serialize, i.e., main memory access latency will be exposed twice. The reason is that both long-latency loads are further apart in the dynamic instruction stream than the ROB size, i.e., the second long-latency load does not reside in the ROB concurrently with the first long-latency load. We conclude from this illustration that in order to accurately model memory-level parallelism, it is important to accurately model the distance between long-latency cache misses. This is achieved in our memory data flow model through cache miss correlation modeling.

For modeling cache miss correlation, we collect cache miss rate statistics per static memory operation—per load/store in an SFG-node—dependent on its *global* cache miss history, not the local per-load/store cache miss history. The global cache miss history is a concatenation of the most recent cache hit/miss outcomes of all the preceding memory references. In the above example where a loop walks

over an array, cache miss correlation allows for making a distinction between the load operation that results in a cache miss and the other load operations that result in cache hits. The global cache miss history for the load miss looks like '0111' where a '0' denotes a cache miss and a '1' denotes a cache hit. The probability for a cache miss given the global cache miss history '0111' equals 100%. The probability for a cache miss equals 0% for the other cache miss histories, '1011', '1101' and '1110'. By doing so, a cache miss will be generated every four loop iterations in the synthetic trace; this matches the original program execution exactly, and captures the MLP in an accurate manner.

An important choice that needs to be made for modeling cache miss correlation is how deep the global cache miss history should be. We consider two implementations. In our first implementation we choose the global cache miss history as deep as the number of preceding loads/stores in the basic block history as determined by the order of the SFG. We will refer to this approach as the *coupled cache miss correlation approach*. This means that in a k-th-order SFG, the hit/miss outcomes for *all* preceding loads and stores in the k-deep basic block history serve as a history for the current memory operation's hit/miss probability. By doing so, we correlate the current load/store hit/miss outcome with the preceding hit/miss outcomes from the k-deep basic block history; we assume $k = 10$ for the coupled approach unless mentioned otherwise. In our second implementation, the *decoupled cache miss correlation approach*, we decouple the global cache miss history from the statistical flow graph by computing the current memory operation's hit/miss probability dependent on the hit/miss history of the n preceding loads and stores, and independent of the basic block history. As will be shown in the evaluation section of this chapter, the decoupled cache miss correlation implementation is slightly less accurate than the coupled approach, but, it needs significantly less memory when collecting the statistical profile and requires less disk space for storing the statistical profile.

We use the cache miss correlation statistics for driving the generation of cache misses when we generate the synthetic trace. When a hit/miss outcome needs to be determined, the global hit/miss outcome generated so far is used to search the cache miss correlation statistic for the given memory operation. The hit/miss probability corresponding to the best matching hit/miss history for the given memory operation is then used for determining whether the memory operation will cause a hit or a miss.

Implementing cache miss correlation in an efficient manner is a challenging task. Our method as detailed above is fairly efficient in terms of additional storage needed in the statistical profile. By implementing the global cache miss history as an array of bits, we only need a few extra bytes per load. Previous work done in SMART by Iyengar et al. [40, 41], however, correlates load misses with fully qualified instructions, as discussed in the previous chapter. In other words (and in terms of SFG terminology), they build cache miss correlation inside the structure of the SFG—they do not simply annotate the SFG with cache miss correlation information as we do; they actually restructure the SFG such that cache miss correlation is embedded in the SFG. As a result of that, they have separate nodes in the SFG in case one of the basic blocks in the basic block history has an instruction that has a different cache miss behavior. This makes the SFG explode—the authors of [40, 41] admit that for some benchmarks they were unable to build a fully qualified SFG in memory. In our work on the other hand, we use the SFG for keeping track of the control flow and use a global cache miss history per load for dealing with cache miss correlation. This leads to a more space-efficient solution in terms of memory usage and storage requirements.

4.3 Read-after-write memory dependences

Out-of-order execution of memory operations is an important source of performance gain in out-of-order microprocessors. The goal is to execute load instructions as soon as possible (as soon as their source operands are ready) provided that read-after-write (RAW) dependences through memory are respected. This allows load instructions to be executed before independent preceding store instructions.

Early execution of loads is achieved in out-of-order microprocessors through two techniques, load bypassing and load forwarding [43]. Load bypassing refers to executing a load earlier than preceding stores; this is possible provided that the load address does not alias with those store addresses. On the other hand, load forwarding allows the load to retrieve its data directly from the store without accessing the memory hierarchy if the load address aliases with a preceding store address—there is a RAW dependence.

Modeling load bypassing and load forwarding can be done in statistical simulation by measuring the *RAW memory dependence* distribution.

This distribution quantifies the probability that a load address aliases with one of the preceding store addresses, and is measured on a per-instruction basis in the context of the SFG. In the synthetic trace, RAW dependences through memory are then marked between the memory operations. During synthetic trace simulation, this information is used to determine the scheduling of memory operations, i.e., to determine whether or not load bypassing or load forwarding is possible.

4.4 Delayed hits

Contemporary microprocessors typically use non-blocking caches [27, 49]. Non-blocking caches allow for overlapping cache misses by putting aside load misses in MSHRs while servicing other load instructions. These overlapped load instructions can also be cache misses, thereby exposing MLP in case these loads access different cache lines, or, exposing so called delayed hits in case these loads access the same cache line. Section 4.2 on cache miss correlation addressed the former; this section concerns the latter.

Prior to this dissertation, statistical simulation frameworks only considered cache hits and misses and they did not model delayed hits, i.e., the modeled latencies are the L1, L2 and main memory access latencies. In a processor with non-blocking caches however, load instructions can see latencies that are different from the L1 access latency, L2 access latency and main memory access latency. Consider for example the case where a load accesses cache line A at time t_{100} and this is a cache miss in L2. If the L2 access latency is twenty cycles, the load finishes execution at time t_{120}. If another load accesses the same cache line at time t_{107} it will then see a load execution latency of thirteen cycles. The latter load then is a *delayed hit* or a *secondary miss*. Current statistical simulation frameworks will consider the delayed hit as a hit and will assign the L1 access latency to this load which is an underestimation of the load's execution latency.

In order to model delayed hits within statistical simulation, we compute the *missed cache line reuse distance*. This is the number of memory references between two memory references accessing the same cache line, of which the first memory reference in the dynamic instruction stream is a cache miss. This is measured per instruction depending on the basic block history (through the SFG). Since an instruction may have multiple missed cache line reuse distances depending on the in-

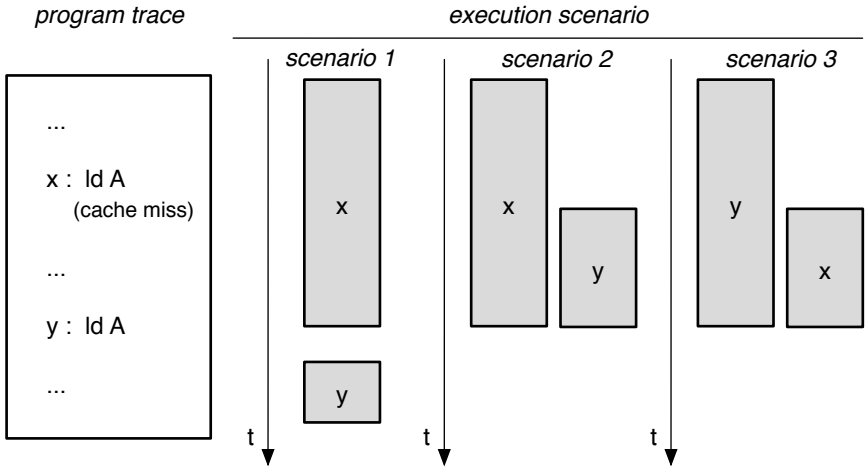

Figure 4.2: Modeling delayed hits in the synthetic trace simulator.

struction's basic block history, we in fact measure a distribution of the missed cache line reuse distance.

We measure the missed cache line reuse distance for both load and store operations; this allows for modeling delayed hits for various cache write policies (write-back and write-through) and cache allocation policies (write-allocate and write non-allocate). An additional optimization that we explore, is to measure the missed cache line reuse distance distribution dependent on the cache miss correlation information. In such way, we are able to more accurately model delayed hits based on global cache miss history information. This was beneficial for the accurate modeling of several benchmarks as will be shown in the evaluation, see Section 4.6.

In order to model delayed hits in the statistical simulation framework, slight modifications need to be made to the synthetic trace simulator. This is illustrated in Figure 4.2. Consider the program trace shown on the left; we have a load miss `x` to cache line A followed by a load hit `y` to the same cache line. There are three possible scenarios that need to be modeled in the synthetic trace simulator:

(1) load `x` has finished its execution when load `y` is issued. Load `y` then gets assigned the L1 access latency. This scenario is accurately modeled in existing statistical simulation frameworks.

(2) load `x` is still executing when load `y` is issued. Load `y` then gets

assigned the remaining execution latency for load `x`.

(3) load `x` is not yet executing when load `y` is issued—this is possible because of out-of-order execution. Load `y` is then turned into a cache miss and thus gets assigned the next cache level's access latency. Load `x` which is issued later on, then gets assigned the remaining execution latency of the resolving load `y`.

The latter two scenarios need special support in the synthetic trace simulator for accurate memory data flow modeling.

4.5 Experimental setup

We use SimpleScalar/Alpha v3.0 [9] in our experiments; we have enhanced the out-of-order simulator in the SimpleScalar Tool Set with a more realistic memory subsystem including MSHRs and store buffers. We use Wattch [8] for estimating energy consumption, which will be used to search for the most energy-efficient microarchitectural configuration in a large design space.

The benchmarks along with their reference inputs used in this study are the SPEC CPU 2000 benchmarks, see Table 4.1. The binaries of these benchmarks are taken from the SimpleScalar website[1]. We consider single (and early) simulation points of one hundred million instructions as determined by SimPoint [64, 67] in all of our experiments. Note that our goal is not to compare against SimPoint. We are just using SimPoint to reduce the simulation time when evaluating the statistical simulation framework: we compare statistical simulation against detailed simulation, and running the detailed simulations is very time-consuming—this is the main motivation for statistical simulation in the first place. We also consider ten-billion-instruction sequences in order to evaluate statistical simulation for longer instruction sequences.

Table 4.2 shows the processor models that we use in this chapter. It shows the baseline configuration along with eight other configurations. These configurations vary in their processor core, branch predictor and memory hierarchy. The reason for considering multiple configurations is to evaluate the accuracy of statistical simulation over multiple points in the design space.

[1] `http://www.simplescalar.com`

benchmark	input	simpoint
ammp	ref	2,130
applu	ref	18
apsi	ref	46
art	ref-110	67
bzip2	program	9
crafty	ref	0
eon	rushmeier	18
equake	ref	194
facerec	ref	136
fma3d	ref	298
galgel	ref	3,150
gap	ref	2,094
gcc	166	99
gzip	graphic	9
lucas	ref	35
mcf	ref	316
mesa	ref	89
mgrid	ref	6
parser	ref	16
perlbmk	makerand	1
sixtrack	ref	82
swim	ref	5
twolf	ref	31
vortex	ref2	57
vpr	route	71
wupwise	ref	584

Table 4.1: The SPEC CPU2000 benchmarks, their reference inputs and the single 100 M-instruction simulation points being used.

	baseline	config 1	config 2
ROB/LSQ	128/32	32/16	32/16
processor width	8	4	4
I-cache	8 KB	8 KB	8 KB
D-cache	16 KB	16 KB	16 KB
L2 cache	1 MB	1 MB	1 MB
latencies (L1/L2/MEM)	2/20/150	2/20/300	2/20/300
entries in I-/D-TLB	32/32	32/64	32/64
hybrid branch predictor	8 K-entry	2 K-entry	8 K-entry
	config 3	config 4	config 5
ROB/LSQ	32/16	32/16	128/64
processor width	4	4	8
I-cache	32 KB	32 KB	8 KB
D-cache	64 KB	64 KB	16 KB
L2 cache	4 MB	4 MB	1 MB
latencies (L1/L2/MEM)	4/30/300	4/30/300	2/20/300
entries in I-/D-TLB	64/128	64/128	32/64
hybrid branch predictor	2 K-entry	8 K-entry	2 K-entry
	config 6	config 7	config 8
ROB/LSQ	128/64	128/64	128/64
processor width	8	8	8
I-cache	8 KB	32 KB	32 KB
D-cache	16 KB	64 KB	64 KB
L2 cache	1 MB	4 MB	4 MB
latencies (L1/L2/MEM)	2/20/300	4/30/300	4/30/300
entries in I-/D-TLB	32/64	64/128	64/128
hybrid branch predictor	8 K-entry	2 K-entry	8 K-entry

Table 4.2: Simulated processor models used for studying the memory data flow modeling.

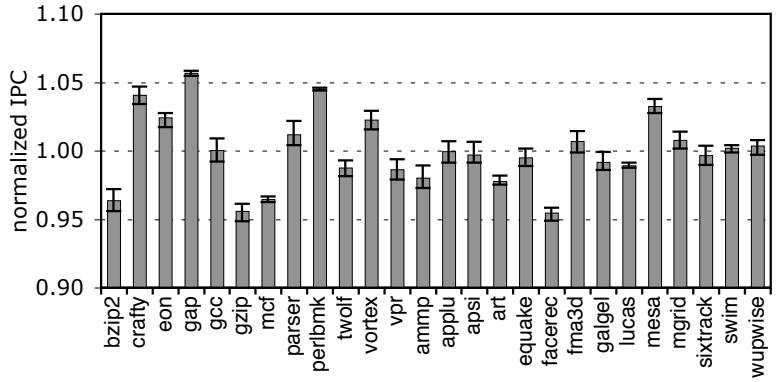

Figure 4.3: Minimum, maximum and average IPC estimates over twenty different runs per SPEC CPU2000 benchmark normalized against the IPC result obtained through detailed simulation; the error bars denote the minimum and maximum IPC value.

4.6 Evaluation

We first quantify the simulation speed of our improved statistical simulation framework. We then quantify the performance prediction accuracy and how it improves through accurate memory data flow modeling. We subsequently measure how well the improved statistical simulation approach can predict performance trends, i.e., we evaluate the relative accuracy and its ability for driving design space explorations. Finally, we also quantify the storage requirements of the statistical profiles and the amount of memory used during statistical profiling. In all experiments we use tenth-order SFGs, unless stated otherwise.

4.6.1 Simulation speed

As stated before, an important feature of statistical simulation is its simulation speed; we speedup the simulation by reducing the dynamic instruction count. Performance characteristics quickly converge to a steady-state value due to the statistical nature of the approach; in the following experiment we show that synthetic traces of one million instructions are sufficient.

For each SPEC CPU2000 benchmark, we have generated twenty synthetic traces of one million instructions. Each of these synthetic

traces was generated from a single statistical profile using different random seeds in the synthetic trace generator. Figure 4.3 shows the minimum, the maximum, and the average IPC value obtained through synthetic simulation, normalized against the IPC value obtained through detailed simulation. A normalized IPC larger or smaller than one reflects an overestimation or underestimation, respectively. From this graph, we observe small gaps between the minimum and maximum normalized IPC values—the minimum and maximum values are shown using error bars in Figure 4.3. In addition, we compute the coefficient of variation (CoV) which is defined as the standard deviation divided by the mean IPC value over those twenty synthetic traces. The CoV we observe is less than 1% for all benchmarks. A CoV $< 1\%$ was also reported in the prior work by Eeckhout et al. [19] which assumed a simple memory data flow model. From this we can conclude that accurate memory data flow modeling has no additional effect on the simulation speed.

4.6.2 Performance prediction accuracy

We now evaluate the performance prediction accuracy for statistical simulation enhanced with memory data flow modeling.

Baseline configuration

Figure 4.4 shows the performance prediction error for the baseline processor configuration for (a) the integer and (b) the floating-point benchmarks, respectively. The IPC prediction error is computed as

$$IPC\ prediction\ error = \frac{IPC_{stat\_sim} - IPC_{det\_sim}}{IPC_{det\_sim}},$$

with IPC_{stat_sim} and IPC_{det_sim} the IPC obtained through statistical simulation and detailed simulation, respectively. A positive error reflects an overestimation whereas a negative error reflects an underestimation. Figure 4.4 shows six bars per benchmark:

- The prior work bar corresponds to previously proposed state-of-the-art statistical simulation approaches—this is the statistical simulation framework including the SFG as described in Chapter 3 and in [19];

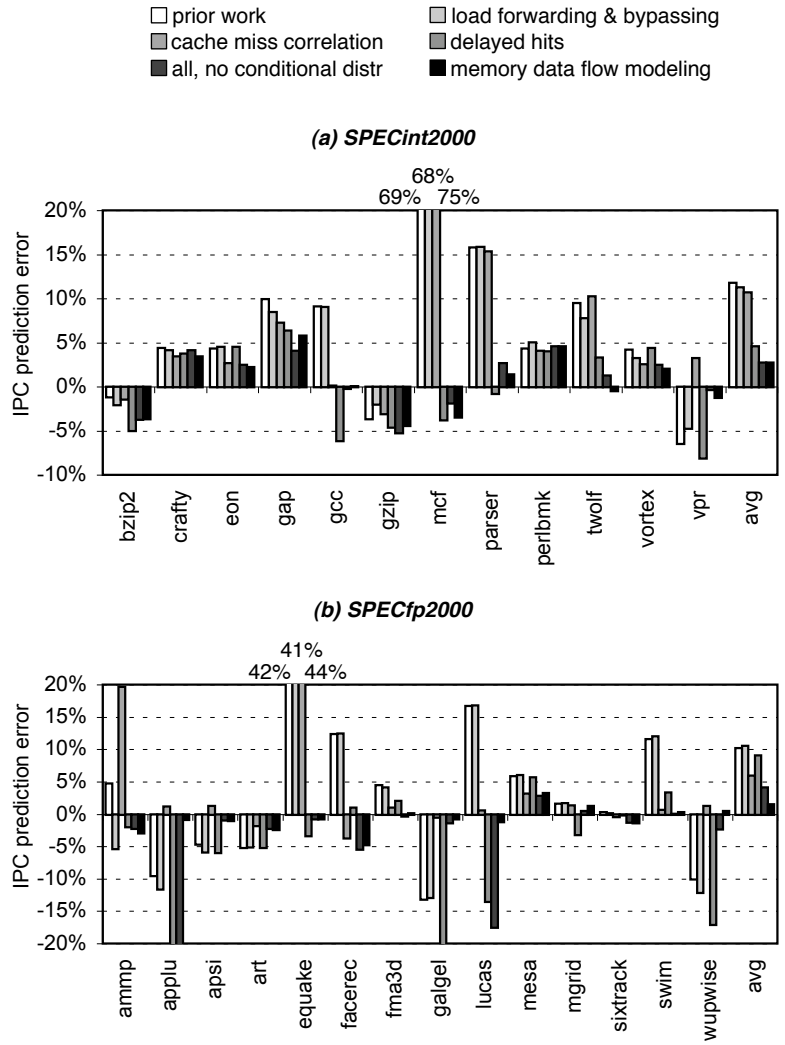

Figure 4.4: IPC prediction error for (a) SPECint2000 and (b) SPECfp2000; evaluating the accuracy of the proposed memory data flow modeling for the baseline processor configuration; these results assume $k = 10$, see discussion on the impact of the order of the SFG k later in this chapter.

- The second bar corresponds to the SFG enhanced with RAW memory dependence modeling;
- The third bar shows the SFG enhanced with cache miss correlation;
- The fourth bar shows the SFG enhanced with delayed hit modeling;
- The fifth bar shows the SFG enhanced with all three enhancements: delayed hit modeling, RAW memory dependence modeling and cache miss correlation, however, the missed cache line reuse distance distribution is not measured dependent on the cache miss correlation information;
- The final bar shows the SFG enhanced with memory data flow modeling. This includes delayed hits, RAW memory dependence modeling and cache miss correlation; in addition, the missed cache line reuse distance distribution is measured dependent on the cache miss correlation information.

Modeling cache miss correlation greatly improves the performance prediction accuracy for a number of benchmarks, see for example gcc, applu, galgel and wupwise. It is interesting to observe that gcc which is known to exhibit complex and irregular memory access patterns, benefits from cache miss correlation. Clearly, cache miss correlation not only enables the accurate modeling of regular cache miss patterns, it is also capable of modeling irregular cache miss patterns. In addition, we do not model the MLP along a speculative path correctly. We send instructions from the correct path in the pipeline as if they were from the speculative path; the amount of MLP along the speculative path may be different from that of the correct path. This may potentially lead to additional inaccuracies.

Modeling delayed hits also decreases the prediction error. The benchmarks that benefit the most from delayed hit modeling are mcf, twolf, ammp, equake, facerec and swim. Additionally, several benchmarks benefit from modeling the cache line reuse distribution dependent on the cache miss correlation information; examples are twolf, applu, and lucas.

The impact of modeling load forwarding and bypassing is rather small. Previous work in statistical simulation, as mentioned before, assumes that a load never aliases with a preceding store. A load can thus

execute as soon as its source operands are available, provided that the addresses of all preceding stores are resolved, which is what the simulation model assumes. Through RAW memory dependence modeling, loads can alias with preceding stores and the IPC prediction through statistical simulation can either increase or decrease. The IPC prediction increases when load instructions only see a one-cycle execution latency getting the store's data from the store buffer or load/store queue; going to the L1 cache takes two cycles in our setup. The IPC prediction decreases when a load has to wait for the data of the aliasing store to be produced. This explains the small changes in IPC prediction error due to RAW memory dependence modeling. Note that our simulation setup does not account for a performance penalty in case the store-load dependence is violated and instructions need to be re-issued—a store-load dependence is never violated because loads have to wait for the addresses of all preceding stores to be resolved. In case a performance penalty is accounted for upon a store-load dependence violation, the importance of RAW memory dependence modeling is likely to increase.

When putting it all together, see the rightmost bars in Figure 4.4, the end result is a highly accurate statistical simulation framework. The average prediction error goes down from 10.9% for prior work[2] to 2.1% in this work—the average errors are computed from absolute errors. The maximum error is observed for gap (5.8%) which is substantially lower than the high errors observed for prior statistical simulation approaches, see for example mcf (69.1%). Note that even without the outlier mcf the average IPC prediction error goes down from 8.7% to 2.0%.

Other processor configurations

The IPC prediction errors discussed above are for the baseline processor configuration given in Table 4.2. We now show results for the other configurations mentioned in Table 4.2. Figure 4.5 on the top shows the average IPC prediction errors for the other eight processor configurations, and compares prior work against statistical simulation enhanced with memory data flow modeling as described in this chapter. We observe that the errors drastically reduce through accurate memory data flow modeling. The average IPC prediction error for previous work varies between 7% and 16% depending on the processor configuration;

[2]Note that this average error is higher than the error reported in [19]; this is because [19] only considered a subset of the SPEC CPU2000 benchmarks.

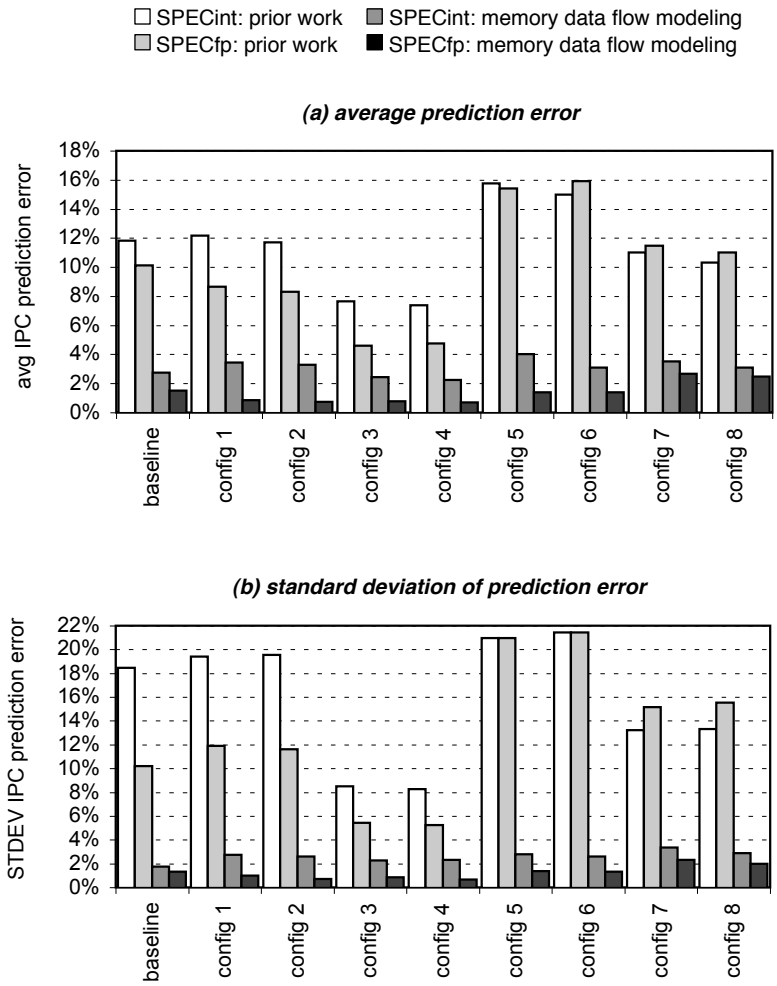

Figure 4.5: (a) Average IPC prediction errors, and (b) standard deviation of IPC prediction errors, for the eight processor configurations from Table 4.2 as obtained through prior work [19] and through enhanced memory data flow modeling.

without mcf, the average error varies between 5% and 13%. With accurate memory data flow modeling, the average error across the SPECint and SPECfp benchmarks is smaller than 4%.

Another interesting observation to be made from Figure 4.5 is that higher errors are observed for wider-resource machines, i.e., the machine configurations with a large 128-entry ROB and a processor width of 8 seem to result in higher prediction errors than the machines with a 32-entry ROB and a processor width of 4 (compare configurations 4 to 8 versus configurations 1 to 4). This can be understood intuitively since the observed parallelism is limited more by program parallelism (instruction-level parallelism and memory-level parallelism) than machine parallelism for wider-resource machines. In such way, modeling inaccuracies becomes more apparent.

Figure 4.5 at the bottom shows the standard deviation of the IPC prediction errors per processor configuration of Table 4.2. A small standard deviation means that the IPC prediction errors for each of the benchmarks are close to the average IPC prediction error for a given processor configuration. With accurate memory data flow modeling the standard deviation across the SPECint and SPECfp benchmarks is much smaller as compared to prior work. Therefore, statistical simulation enhanced with accurate memory data flow modeling not only improves the average error, but it also shows less variation in the prediction errors across the benchmarks. In other words, accurate memory data flow modeling resolves the issues with the outliers in prior work.

Comparing coupled and decoupled cache miss correlation approach

Recall there are two approaches to modeling cache miss correlation, namely a coupled and a decoupled approach. The coupled approach takes a global hit/miss history that is as long as the number of all loads and stores for all k most recent basic blocks with k being the order of the SFG. The decoupled approach takes a global hit/miss history of n loads and stores. Figure 4.6 compares the coupled approach with $k = 10$ versus the decoupled approach with $k = 1$ and $n = 50$ [3]. The decoupled approach is only slightly less accurate than the coupled approach;

[3] On average, the SPEC CPU2000 benchmarks contain 4.4 memory references per basic block. Thus, the history of memory references in a tenth-order SFG corresponds to a history of $n = 4.4 \times 10$ memory references on average. Therefore, the decoupled approach with a history of length $n = 50$ closely approximates the coupled approach with a tenth-order SFG. Another workload may require to adjust the parameter n.

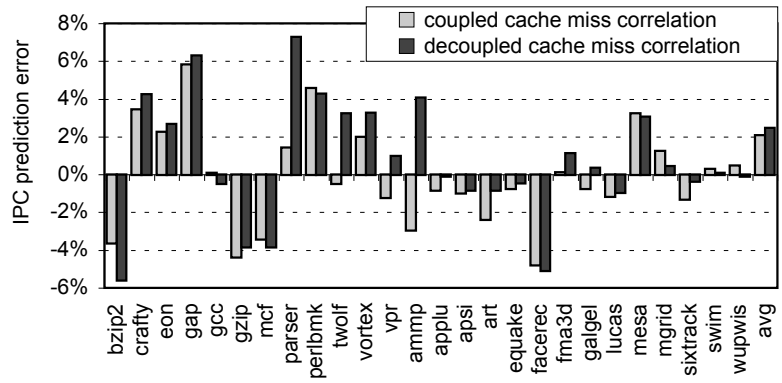

Figure 4.6: Comparing coupled versus decoupled cache miss correlation modeling in terms of accuracy for the baseline configuration.

the average IPC prediction error increases from 2.1% to 2.5%. As will become clear later on in this chapter, the benefit of the decoupled approach is the reduced memory needs and the reduced disk space requirements as compared to the coupled approach.

Impact of global hit/miss history length

We now evaluate the impact on accuracy of the length of the hit/miss history as used for modeling cache miss correlation for both the coupled and the decoupled approach.

Figure 4.7 quantifies the impact on IPC prediction error of the SFG's order k for the coupled approach. This graph shows the average IPC prediction error for different values of k. We observe that, as expected, the IPC prediction error decreases with increasing k. The error stabilizes between 2.3% and 2.1% past $k = 8$ and $k = 10$, respectively.

Figure 4.8 shows the sensitivity of the decoupled approach to the hit/miss history length. For both the first-order and third-order SFGs, the performance prediction error stabilizes around 2.6% and 2.1%, respectively, as soon as the hit/miss history length n is larger than 40. The prediction error for the third-order SFG is only marginally smaller than the prediction error for the first-order SFG.

There are basically two reasons why the accuracy improves with increasing k in the case of the coupled approach. For one, as stated in [19], a higher-order SFG incorporates path information into the sta-

Figure 4.7: Average IPC prediction error as a function of the SFG's order k for the coupled cache miss correlation approach.

Figure 4.8: Average IPC prediction error as a function of the global cache hit/miss history length for the decoupled cache miss correlation approach; the SFG's order remains unchanged while varying the history length.

tistical profile. Benchmarks for which program characteristics correlate well with this path information benefit from this. However, this effect is rather limited as discussed in [19]. The second and more important reason is that a higher-order SFG also implies that a longer global cache miss history is taken into account for modeling the cache miss correlation as explained in Section 4.2.

CPI breakdown

Figure 4.9 shows the CPI breakdown as obtained through detailed simulation (DS) and statistical simulation (SS) for (a) the integer and (b) the floating-point benchmarks, respectively. The CPI stack can be broken up into its base CPI, branch mispredict CPI, I-cache miss CPI and D-cache miss CPI. The base CPI component is obtained through simulation assuming perfect branch prediction and perfect caches, i.e., no misses occur. The branch predictor CPI component is obtained assuming perfect caches and a realistic branch predictor. The I-cache and D-cache CPI components consider realistic I-cache and D-cache, respectively, and both components assume everything else as perfect. We clearly observe that statistical simulation is indeed able to accurately track the various CPI components. In other words, the accurate performance modeling is not a result of compensating modeling errors in the various CPI components.

Long instruction sequences

All of the above results were done on relatively short one-hundred-million-instruction sequences selected using SimPoint. Figure 4.10 shows the IPC prediction error for the baseline configuration on ten-billion-instruction sequences (after skipping the first one billion instructions which is mostly initialization code). These results show that statistical simulation with enhanced memory data flow modeling is also very accurate for long instruction sequences with errors varying between -2.4% and 3.8%, and an average (absolute) error of 1.6%. Moreover, the synthetic trace counts one million instructions whereas the original trace consists of ten billion instructions. This is a four-order of magnitude reduction in simulation time.

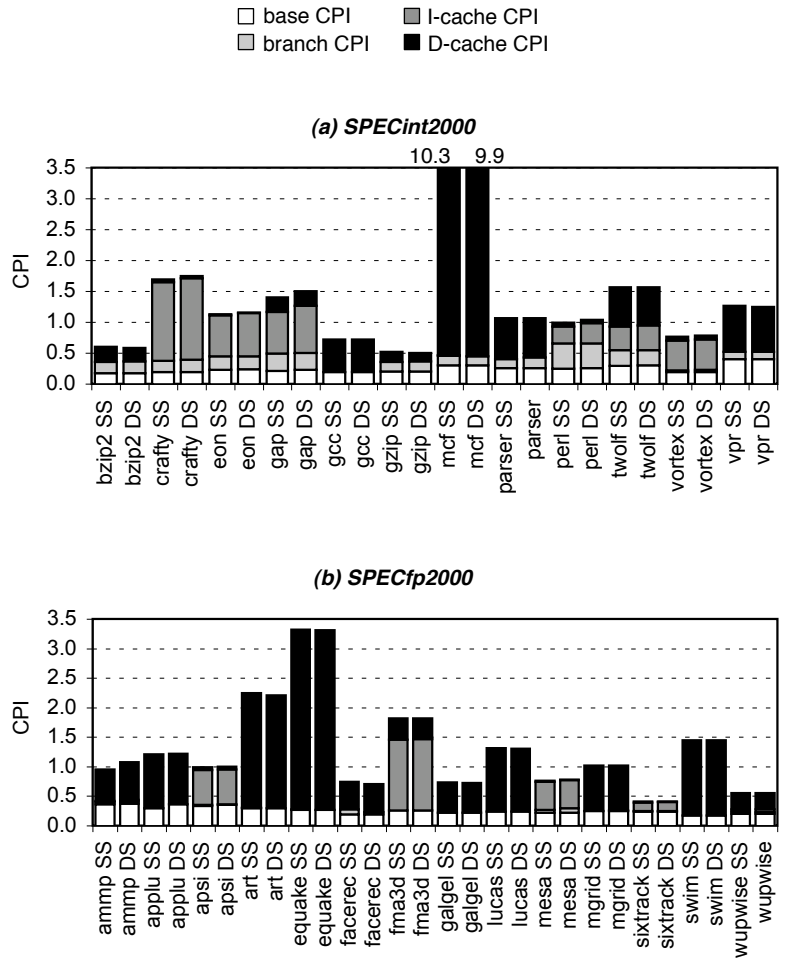

Figure 4.9: CPI breakdown for detailed simulation (DS) and statistical simulation (SS) for both (a) SPECint2000 and (b) SPECfp2000 benchmarks.

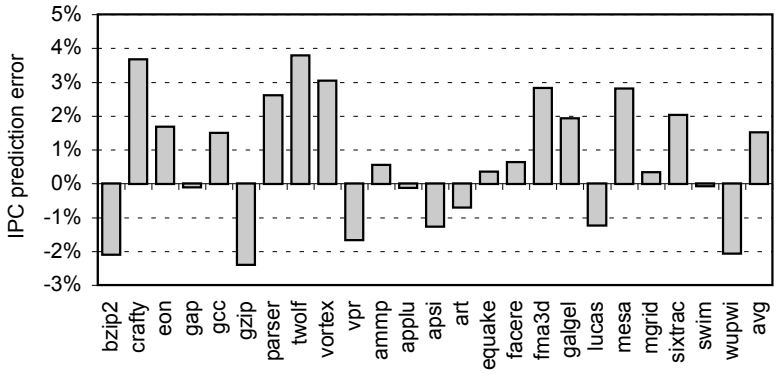

Figure 4.10: IPC prediction errors for ten-billion-instruction sequences.

4.6.3 Sensitivity to cache hierarchy parameters

We now evaluate the ability of the proposed statistical memory data flow model to track performance differences across a wide variety of cache hierarchy designs. For each experiment we present the average performance over all benchmarks, and in addition, we show three individual benchmarks—we retrieved similar results for other benchmarks. Each graph shows the IPC for both detailed simulation and statistical simulation enhanced with memory data flow modeling. We assume processor configuration 7 in all of these experiments.

In our first experiment we vary the *cache line size*. The graphs in Figure 4.11 show the IPC as a function of four cache line sizes: 32 B, 64 B, 128 B and 256 B.

The second experiment studies the accuracy of statistical simulation when changing the *cache line updating policy*. We consider two cache write policies: write-back (WB) and write-through (WT), and two cache allocation policies: write-allocate (WA) and write non-allocate (WNA), see Figure 4.12.

Figure 4.13 shows IPC as a function of the *MSHR configuration*. Both the number of MSHR entries and the number of targets per entry are varied.

Finally, we study the impact of *the size of the store buffer*. Figure 4.14 shows IPC as a function of the number of entries in the store buffer. The store buffer holds completed stores that still need to be retired, i.e., the value still needs to be written to the memory hierarchy although the

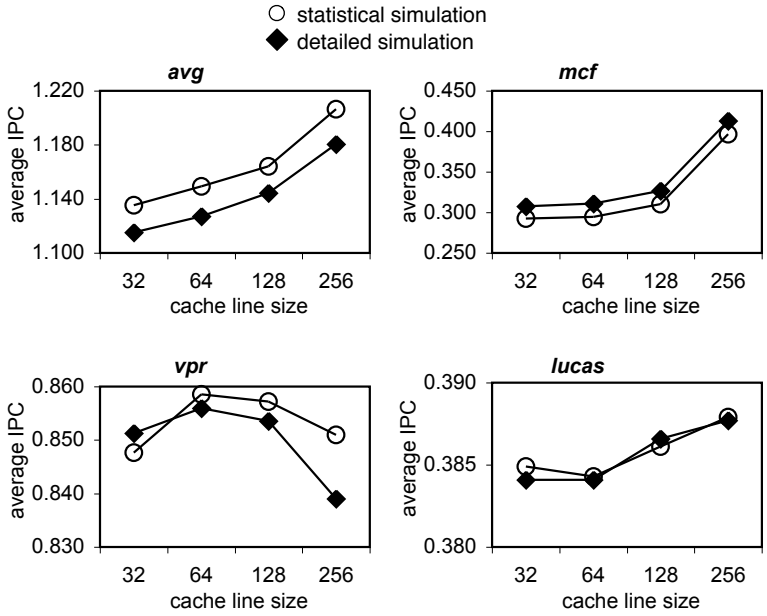

Figure 4.11: Estimating IPC as a function of cache line size.

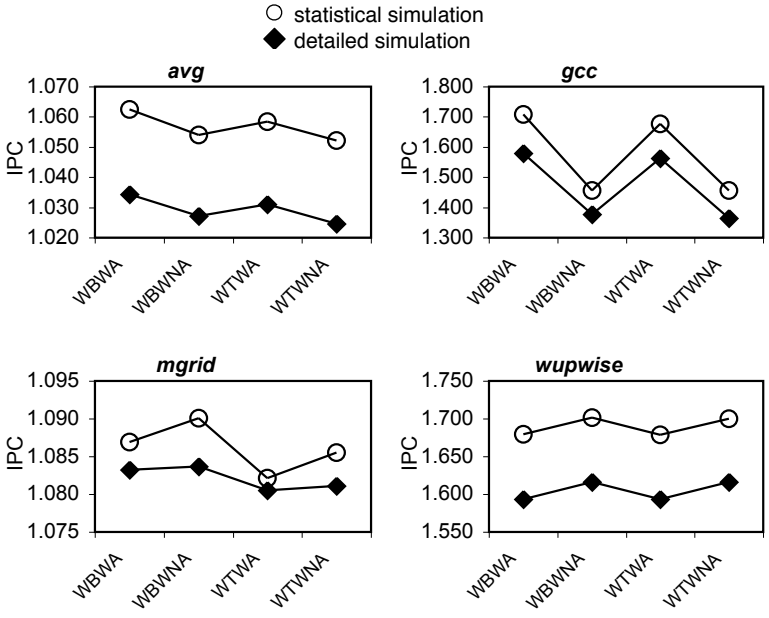

Figure 4.12: Estimating IPC under four cache line updating policies.

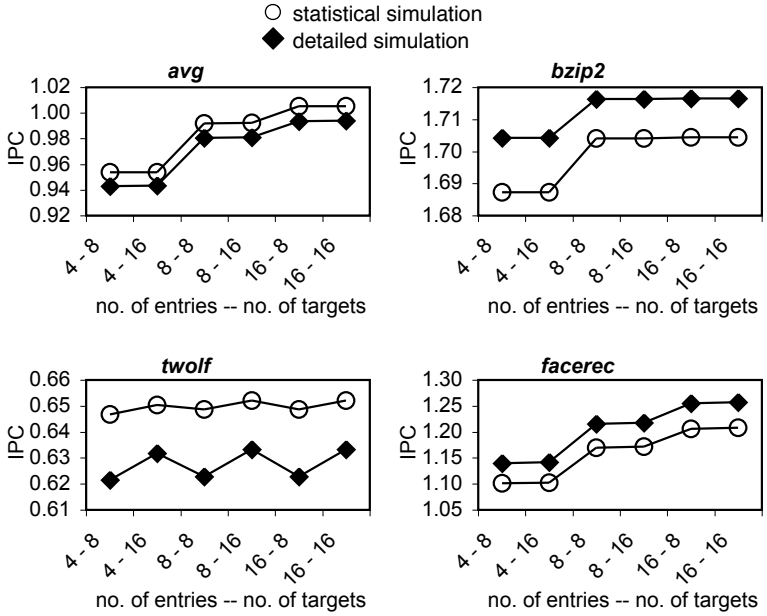

Figure 4.13: Varying the MSHR configuration.

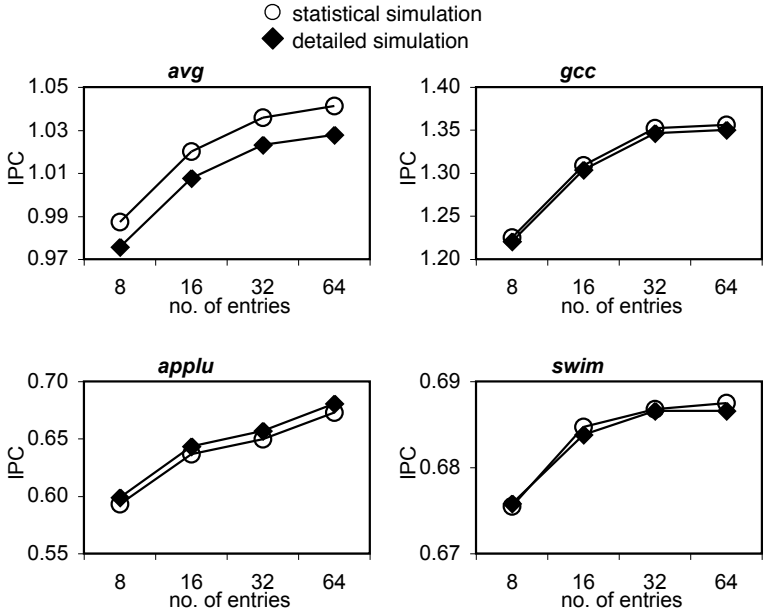

Figure 4.14: Varying the number of store buffer entries.

store already is architecturally completed.

We conclude that in spite of the (small) absolute IPC prediction error, statistical simulation accurately tracks the small relative performance differences.

4.6.4 Trend prediction

For microprocessor design studies, the ability to predict performance trends is very important. A computer designer typically wants to find the *knee* of the performance curve, i.e., the point where the performance curve starts to flatten. Note that this is particularly true during the early stages of the design cycle. To quantify this we compute the IPC as obtained from detailed simulation runs and compare that against the IPC as obtained from statistical simulation. We vary a number of microarchitectural parameters and compute how well statistical simulation tracks the real simulation data. The microarchitectural parameters that we consider here in this section are specifically targeted towards the memory hierarchy, namely the L1 D-cache access latency, the L2 cache access latency and the load/store queue size. Note that these performance trends can be estimated through statistical simulation from a single statistical profile. We vary the L1 D-cache access latency from 2 to 6 cycles, the L2 access latency from 10 to 30 cycles and the load/store queue size from 16 to 256 entries.

Our metric here is the relative speedup prediction error which is defined as follows:

$$RE = \frac{IPC_{B,SS}/IPC_{A,SS} - IPC_{B,DS}/IPC_{A,DS}}{IPC_{B,DS}/IPC_{A,DS}},$$

with SS and DS standing for statistical simulation versus detailed simulation, respectively, and A and B being two processor configurations. Figure 4.15 shows the maximum relative error that we observe along the three above mentioned microarchitectural parameters for the SPEC CPU2000 benchmarks. We observe that statistical simulation without memory data flow is fairly accurate. However, when memory data flow is enabled even smaller relative prediction errors are observed. We conclude that accurate memory data flow allows for more faithful performance trend predictions.

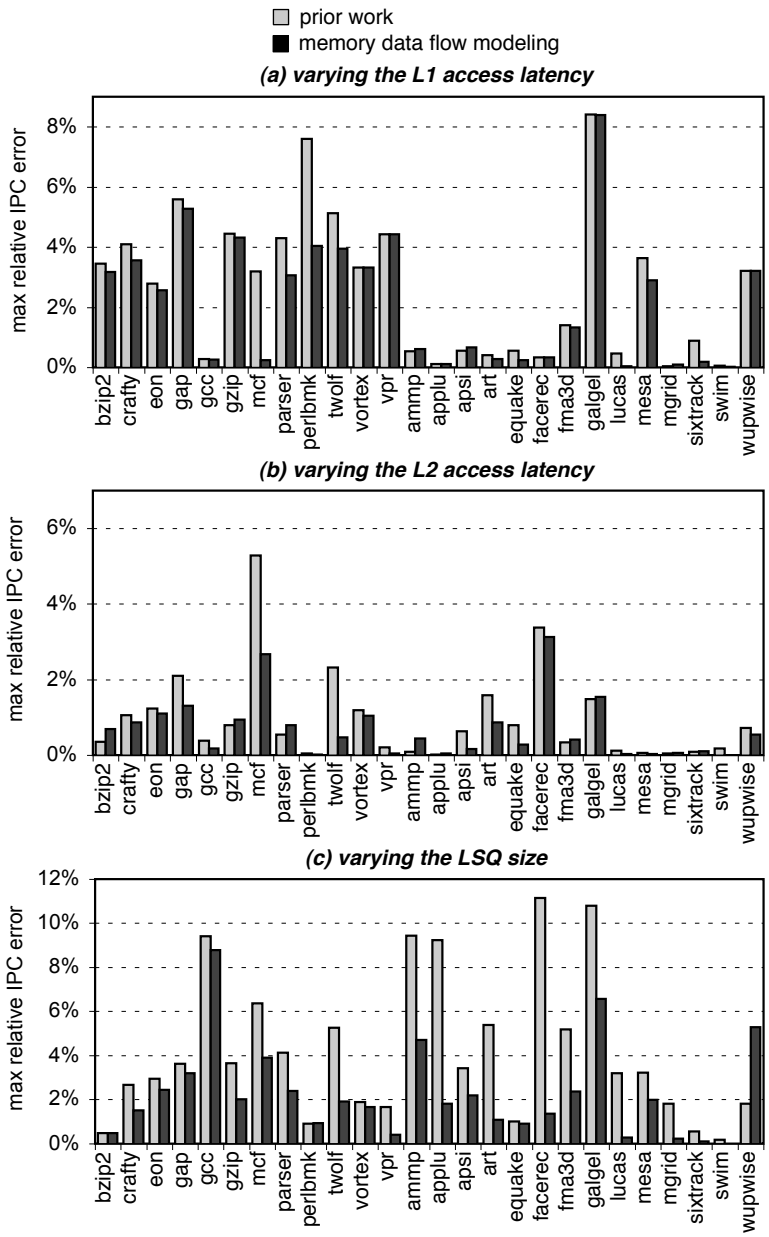

Figure 4.15: Maximum relative error is shown while varying the L1 D-cache access latency (on the top), the L2 access latency (in the middle), and the load/store queue size (at the bottom). This is for processor configuration 7, see Table 4.2.

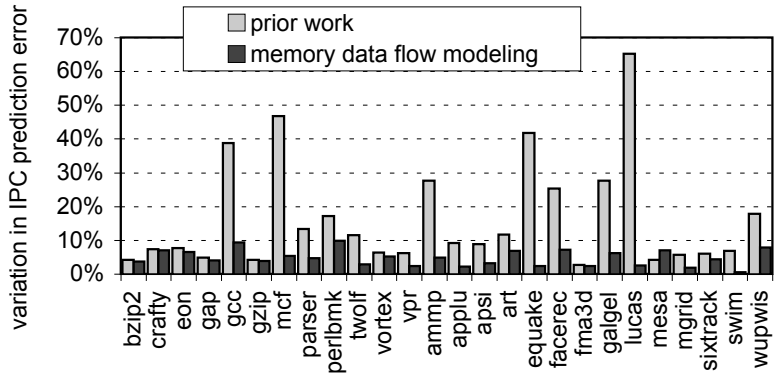

Figure 4.16: Maximum variation in IPC prediction error observed across the eight processor configurations from Table 4.2.

4.6.5 Error distribution across the design space

As observed from Figure 4.5 where the IPC prediction error was shown for eight processor configurations, the IPC prediction error varies over the design space. Figure 4.16 quantifies the maximum variation observed in IPC prediction errors across these eight processor configurations. This maximum variation is defined as the maximum IPC prediction error difference between two configurations:

$$variation_{max} = \mid \max(IPC_{error,i}) - \min(IPC_{error,i}) \mid,$$

over all configurations i.

Prior work in statistical simulation seems to be susceptible to large variations in IPC prediction error across the design space, see for example gcc (39%), mcf (47%), equake (42%) and lucas (65%). For statistical simulation with enhanced memory data flow modeling this maximum variation in IPC prediction error drops substantially below 10% for all benchmarks. This maximum variation across processor configurations is an important metric for design space explorations because it quantifies the variation in IPC prediction error across the design space. Too large variations can lead to suboptimal design decisions when statistical simulation is used for exploring the design space. Clearly, accurate memory data flow modeling helps statistical simulation to yield a smaller variation in IPC prediction errors across the design space.

4.6.6 Design space exploration

We now evaluate the ability of statistical simulation to identify the optimal design point in a given design space. We define the optimal design point as the design point with the minimum energy-delay-square product (ED^{2}P), i.e., $ED^2P = CPI^2 \times EPI$, which is an appropriate metric for quantifying energy-efficiency in high-end server processors [7]. The design space is built up by varying the ROB size from 8 to 256 entries; the LSQ is varied from 4 to 256 entries (with the additional constraint that the LSQ is never larger than the ROB); and the processor width (decode, dispatch, issue and commit) is varied from 2 to 8. Note that all of these statistical simulations are run from a single statistical profile using one million instruction synthetic traces; this is more than a factor one hundred faster than the detailed simulation using one hundred million simulation points in our experimental setup. The optimal design points identified through statistical simulation with enhanced memory data flow modeling exactly matched the optimal design points identified through detailed simulation for 20 out of the 26 benchmarks; for the other 6 benchmarks, the optimal design point identified through statistical simulation was within 3% of the optimum identified through detailed simulation. This is far more accurate than what is obtained through statistical simulation without the enhanced memory data flow modeling, as used in prior work. Prior work gets off the optimal design point for eleven benchmarks and the deficiency is even fairly large for four of these benchmarks: art (4.5%), mcf (5.3%), bzip2 (5.7%) and equake (14.6%). We conclude that statistical simulation enhanced with accurate data flow modeling is highly accurate (and significantly more accurate than prior work) in identifying a region of (near-)optimal design points in a large design space.

4.6.7 Storage requirements

Figure 4.17 on the left shows the average size of the (compressed) statistical profiles in MB as a function of the order k of the SFG for the coupled cache miss correlation approach; these are average numbers over all the benchmarks. For $k = 8$ and $k = 10$, the average statistical profile requires 4.6 MB and 5.8 MB of disk space, respectively. The storage requirements for the decoupled approach are smaller, see Figure 4.17 on the right. For $n = 40$ and $k = 3$ (which achieves similar accuracy as $k = 10$ for the coupled approach), the average statistical

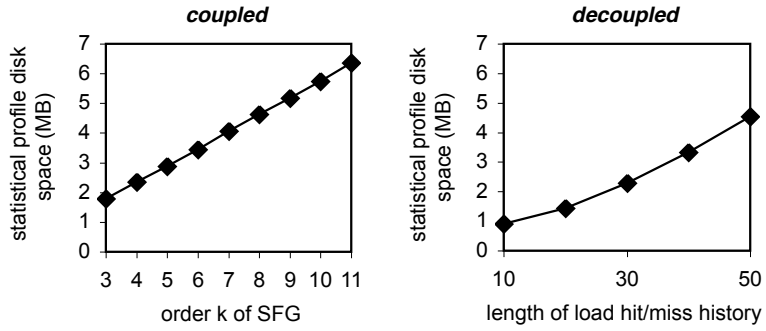

Figure 4.17: Average disk space requirements (in MB) for storing the compressed statistical profiles for both the coupled and the decoupled cache miss correlation approach; the coupled approach graph shows the disk space requirements as a function of the order k of the SFG, the decoupled approach graph shows the disk space requirements as a function of the hit/miss history length n.

profile size equals 3.3 MB. These results are obtained using the one-hundred-million-instruction simulation points considered in this dissertation. For the ten-billion-instruction sequence, the statistical profile is only 17 MB on average per benchmark. We thus conclude that the storage requirements for accurate memory data flow modeling are fairly small.

4.6.8 Memory usage

Memory usage during the statistical profiling step is another important issue. We already referred to the work done by Iyengar et al. [40, 41] in which it was impossible to build the SFG in memory for particular programs during statistical profiling. Figure 4.18 shows the average memory usage during statistical profiling for both the coupled and the decoupled approach. For the coupled cache miss correlation approach with $k = 10$ and the one-hundred-million-instruction simulation points, the average amount of memory used on a 64-bit AMD machine was 897 MB; some benchmarks use even more memory, up to 7.9 GB (equake), 3.5 GB (ammp) and 3.1 GB (crafty). The decoupled cache miss correlation approach with $n = 40$ and $k = 3$ on the other hand, uses significantly less memory, on average 380 MB which is a $2.4\times$ reduction; the extremes were also substantially smaller—the

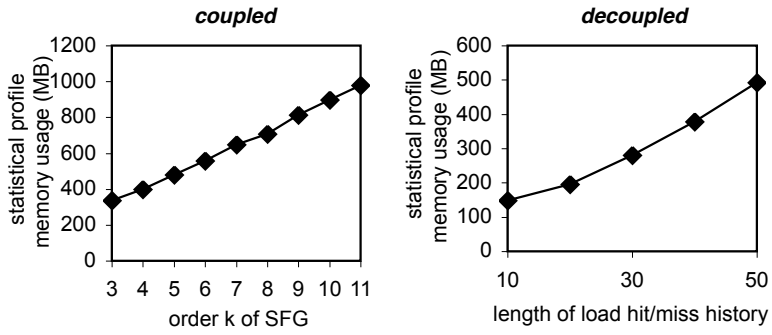

Figure 4.18: Average memory usage (in MB) while computing the statistical profiles for both the coupled and the decoupled cache miss correlation approach; the coupled approach graph shows the memory usage as a function of the order k of the SFG, the decoupled approach graph shows the memory usage as a function of the hit/miss history length n.

maximum was observed for ammp with 1.0 GB. For the ten-billion-instruction sequences, we were able to run the statistical profiling step using the decoupled cache miss correlation approach on our 64-bit AMD machine with 8 GB of physical memory. However, we were unable to run the statistical profiling step using the coupled cache miss correlation approach.

4.7 Summary

In this chapter we have introduced accurate memory data flow modeling in statistical simulation. Accurate memory data flow modeling involves the modeling of: (i) cache miss correlation, making all memory characteristics dependent on a global cache miss history, (ii) load bypassing and load forwarding through RAW memory dependences, and (iii) delayed hits through missed cache line reuse distances.

Our experimental results using the SPEC CPU2000 benchmarks show that significant reductions in IPC prediction errors are obtained by more accurately modeling memory data flow characteristics. For our baseline configuration we reported a reduction in average IPC prediction error from 10.9% down to 2.1%. We also showed that the variation in IPC prediction errors across different microarchitectures is significantly smaller when memory data flow is modeled. In addition,

performance trends are predicted more accurately which is extremely important for design space exploration purposes.

Furthermore, we showed that accurate memory data flow modeling has little or no impact on simulation speed. Synthetic traces of one million instructions are still sufficient for obtaining converged performance estimates, while obtaining a simulation speedup of up to four orders of magnitude. A possible disadvantage are the memory requirements during the profiling phase, however, we have shown a technique to address this issue, i.e., decoupling the cache miss correlation from the SFG drastically reduces the memory space requirements without sacrificing accuracy too much.

Chapter 5

Chip-Multiprocessor Design Space Exploration through Statistical Simulation

We used to think that if we knew one, we knew two, because one and one are two. We are finding that we must learn a great deal more about "and".
Arthur Stanley Eddington

As mentioned in the introduction, the cores of a chip-multiprocessor share resources such as last-level caches, off-chip bandwidth, interconnection network and main memory, resulting in resource contention between co-executing threads. The conflict behavior in shared resources strongly depends on the microarchitectural configuration: conflicts in the shared resources may cause some threads to run slower than others, leading to different relative progress rates for the co-executing threads. Changing which phases of the threads that run together in turn affects the conflict behavior in the shared resources. This tight performance entanglement between co-executing threads and the microarchitecture, makes it hard to model performance in multicore simulation.

In this chapter we enhance the statistical simulation framework from Chapters 3 and 4 such that it can deal with the tight performance entanglement between co-executing threads. We will focus on chip-multiprocessors running multiprogram workloads; Nussbaum and Smith [60] and Hughes and Li [38] show how inter-thread synchronization can be modeled in the case of multithreaded workloads.

Because the number of conflicts cannot be determined at profile

time, we must model a shared cache access independently of the cache associativity parameter and then infer the conflict behavior when simulating the synthetic traces. Our approach models cache accesses independently of the associativity, and moreover, independently of the number of sets in the cache. An additional benefit of microarchitecture-independent cache modeling is that more cache design points can be studied from a single statistical profile, which improves the applicability of statistical simulation for design space explorations.

Furthermore, we model DRAM behavior more realistically. Prior work assumes a simple DRAM model, returning a fixed amount of cycles as its access latency, which differs from real main memory systems. Modern DRAM modules typically implement an open page policy, allowing for shorter access latencies upon a row hit. In addition, DRAM modules are composed of a number of independent banks, allowing accesses to overlap providing bank-level parallelism and thus better performance. As a result a main memory access has a variable latency, depending on a bank/row hit/miss. Therefore, statistical simulation requires enhancements in memory access modeling as well.

5.1 Shared resource modeling

Chapter 3 describes how statistical simulation models the memory address stream through microarchitecture-dependent cache miss statistics; the statistical profile captures the cache miss rates of the various levels in the cache hierarchy. The synthetic instructions are labeled with cache miss information, indicating whether the memory access results in an L1 hit, L2 hit or L2 miss. Although this is sufficient for the statistical simulation of single-core processors, it is inadequate for modeling chip-multiprocessors with shared resources in the memory hierarchy. As mentioned before, the level of interaction between co-executing programs is greatly affected by the microarchitecture and by which specific program phases that run together. Therefore, cache miss rates profiled from single-threaded execution are unable to capture conflict behavior in shared resources of a chip-multiprocessor when co-executing multiple programs. Our aim is to model memory access behavior in the synthetic traces independently of the memory hierarchy, such that conflict behavior among co-executing programs can be derived during the simulation of the synthetic traces.

Another issue we face when modeling resource sharing in statistical

simulation is the relative progress of co-executing threads. For example, the number of conflict misses in a shared last-level cache may vary depending on which phases of the threads run together. Therefore, statistical simulation should not only accurately predict the per-thread performance, but it must also exhibit the same phase behavior as the original thread. In other words, statistical simulation must accurately capture the time-varying execution behavior.

5.1.1 Profiling memory address stream characteristics

Modeling memory address stream locality behavior requires that we model the correlation between individual memory accesses. In the discussion on cache miss correlation in Chapter 4 we found that *inter-memory reference correlation* is necessary for accurately modeling memory-level parallelism as well as delayed hits in statistical simulation. The previous chapter models inter-memory reference correlation through correlating hit/miss histories, which depends on the cache configuration considered when recording the statistical profile. Instead, we now use the notion of memory location reuse distance, which is a cache hierarchy independent program characteristic. The *reuse distance* is defined as the number of memory references between two references to the same (*cache-line-sized*) memory location, see Table 5.1. The reuse distance differs from the (LRU) stack depth in that the former counts all memory references, whereas the latter only counts unique memory references.

We compute a distribution of the reuse distance for each memory access in the SFG. We do this for the instruction addresses, as well as for the load's and store's effective addresses. The reuse distance distribution thus captures the temporal locality in the memory address stream. We measure this distribution dependent on the reuse distances of the 50 prior memory references[1] to model memory reference locality in a decoupled way. We measure the history of reuse distances in buckets (of size power of 2) to limit the size of the history of reuse distance distributions that need to be stored as part of the statistical profile. For the same reason we also limit the maximum reuse distance to 4096 memory

[1]See Figure 4.8 and the discussion on the coupled versus the decoupled approach in Chapter 4.

memory reference	A	B	C	D	E	A	A	C	C	E	B	D
referenced cache set	0	0	1	1	0	0	0	1	1	0	0	1
reuse distance	–	–	–	–	–	4	0	4	0	4	8	7
per-set stack depth	–	–	–	–	–	2	0	1	0	1	2	1
row stack depth	–	–	–	–	–	4	0	3	0	2	4	4

Table 5.1: Illustrating the reuse distance and the stack depth for a short memory trace; assuming a two-way set associative cache with two sets {A,C,E,...} and {B,D,...}. A reference hits in a two-way set-associative cache if the per-set stack depth is smaller than two.

references[2].

Furthermore, we keep track of the *virtual memory addresses* that each memory access (instruction pointer and load/store address) touches and how frequently it is touched. Measuring the virtual memory address distribution dependent on the aforementioned reuse distance history models correlation among memory references. Dependent on the virtual memory address, we record three additional memory address stream characteristics, namely the distributions of the *stack depth* for the L1 cache, L2 cache[3] and main memory.

The stack depth for main memory is computed as the number of unique DRAM row accesses made since the last reference to that same DRAM row, assuming a single-bank DRAM design—we will consider multibank DRAM configurations later in this chapter. Similarly, the stack depths for the L1 and L2 caches are computed as the number of unique cache block accesses (per set) since the last reference to that same cache block. Table 5.1 shows the per-set stack depth assuming a set associative cache (with two sets), and the row stack depth assuming a single-bank DRAM design.

For computing the stack depths for the L1 and L2 caches, we assume the largest L1 and L2 cache one may be potentially interested in during design space exploration. The maximum stack depth kept track of during profiling is a, which is the associativity of the largest cache of

[2]The reuse distance between a load miss and a delayed load hit is not larger than the number of ROB entries. On the other hand, the reuse distance between a store miss and a later load access to the same cache line may be larger than the ROB size, because a store does not wait for the miss to be resolved before committing. We measured that the probability of a delayed hit per reuse distance is negligible for reuse distances larger than 4096.

[3]Throughout this chapter we refer to the shared cache as the L2 cache; extending our framework to model shared L3 caches is straightforward.

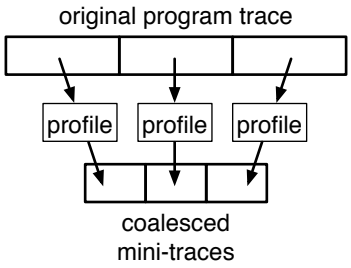

Figure 5.1: Modeling time-varying behavior by dividing the original program trace into intervals.

interest. Accessing the stack at depth a means that a miss occurred in the largest cache of interest, i.e., there are a unique references between the current and the previous access to the given cache line. The stack depth profile can be used to estimate cache miss rates for caches that are smaller than the largest cache of interest, i.e. the cache statistics are modeled independently on the microarchitecture. In particular, all accesses to a stack depth equal to or larger than a will be cache misses in an a-way set-associative cache.

Generating the synthetic traces is done as described in Chapter 3, except that for loads and stores as well as for all instruction addresses, we also determine the reuse distance of the memory location being accessed, their virtual memory address, and their stack depths for the L1 and L2 caches as well as for main memory.

5.1.2 Modeling time-varying execution behavior

A critical issue to the accuracy of statistical simulation for modeling CMP performance is that the synthetic trace has to capture its time-varying execution behavior. The reason is that overall performance is affected by the phase behavior of the co-executing programs. Moreover, the relative progress of a program is affected by the conflict behavior in the shared resources. For example, extra cache misses induced by cache sharing may slow down a program's execution. A program running relatively slow because of cache sharing may result in different program phases co-executing with the other program(s), which in turn may result in different cache sharing behavior, and thus faster or slower relative progress.

Figure 5.1 illustrates how we model time-varying behavior in sta-

tistical simulation. We divide the entire program trace into a number of instruction intervals; an instruction interval is a sequence of consecutive instructions in the dynamic instruction stream. We then collect a statistical profile per instruction interval and generate a synthetic mini-trace. Coalescing these mini-traces yields the overall synthetic trace. Figure 5.2 shows that the synthetic trace exhibits the same phase behavior as the original program trace; this is for equake, we have obtained similar results for the other benchmarks.

The importance of modeling a program's time-varying behavior is further illustrated in Figure 5.3. The six graphs show *relative progress graphs* for two programs co-executing on a multicore processor: equake-wupwise, fma3d-bzip2, gcc-parser, equake-sixtrack, mcf-lucas and galgel-mgrid. A point (x, y) on a relative progress curve denotes that the first program has executed x instructions and the second program has executed y instructions. In other words, a gentle slope denotes that the first program makes fast relative progress compared to the second program, a steep slope denotes that the first program makes slow relative progress compared to the second program. All graphs in Figure 5.3 demonstrate the importance of modeling a program's time-varying behavior. Without time-varying behavior modeling, statistical simulation is unable to track relative progress rates, which leads to inaccurate multicore processor performance predictions, see also Figure 5.4. The reason for this inaccuracy is that very different phases are co-executed under statistical simulation compared to detailed simulation. If a program's time-varying execution behavior is modeled on the other hand, statistical simulation is capable of accurately tracking relative progress rates, which yields substantially more accurate multicore performance predictions. The important insight here is that modeling the large-scale time-varying behavior (at the granularity of millions of instructions) in statistical simulation does not attribute to the accuracy for single-core processor performance estimation, but it does have a substantial impact on the accuracy when co-executing programs on a multicore processor.

Other simulation speedup techniques which target multicore processors with shared resources must also model the time-varying execution behavior, i.e., determine which program phases that run together. For example, Van Biesbrouck et al. [74, 75, 76] propose the co-phase matrix for guiding sampled simultaneous multithreading (SMT) processor simulation running multiprogram workloads. The idea of the co-phase matrix is to keep track of the relative progress of the programs on a per-

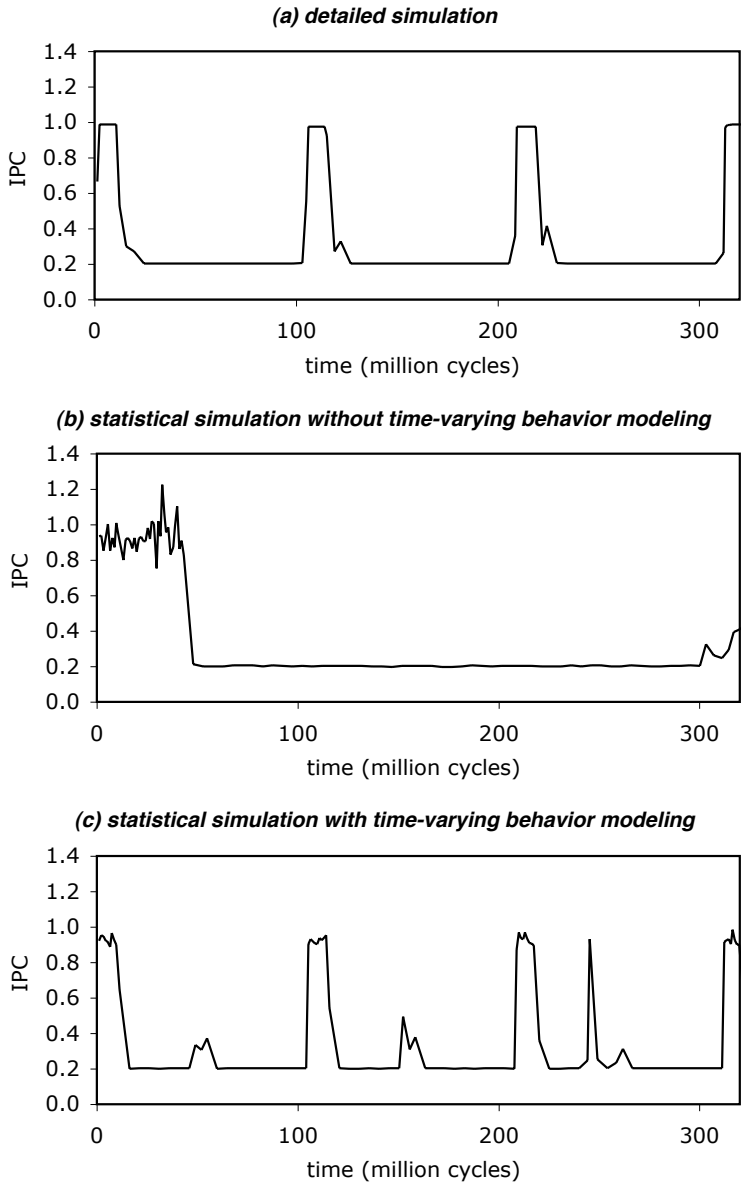

Figure 5.2: Dividing the original program trace into intervals captures the program phases; these graphs are for equake.

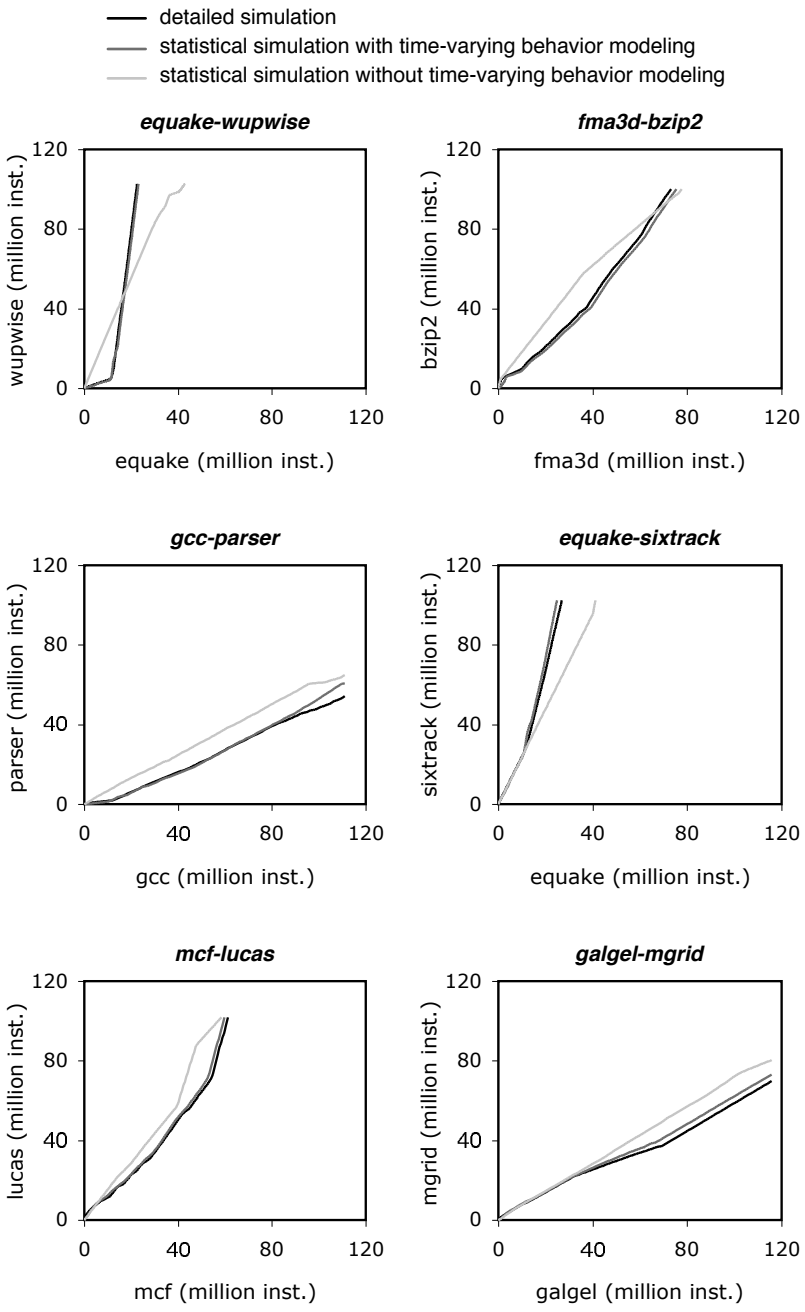

Figure 5.3: Relative progress graphs illustrating the importance of time-varying modeling for six two-program mixes.

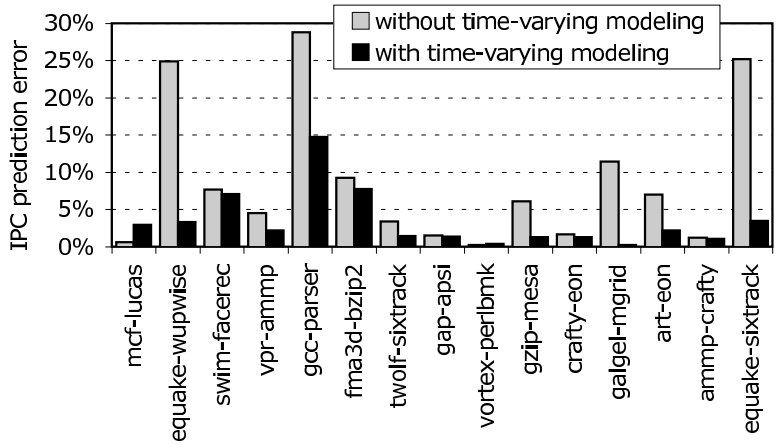

Figure 5.4: Prediction error through statistical simulation with and without modeling a program's time-varying behavior.

phase basis when executed together. By doing so, co-phases need to be simulated only once; the performance of recurring co-phase executions can then simply be read from the co-phase matrix, which speeds up simulation.

5.1.3 Virtual address to physical address translation during synthetic trace simulation

The virtual addresses that appear in the synthetic traces need to be translated into physical addresses during statistical simulation in order to accurately model conflict behavior in physically indexed caches and main memory—the L2/L3 caches are typically physically indexed, whereas the L1 is often virtually indexed to speedup the L1 access time. A naive solution would simply employ the first-come-first-served strategy in statistical simulation as done under detailed simulation, i.e., the next available physical memory page is allocated when a new virtual address page is touched (bump pointer allocation). However, this leads to inaccurate modeling. The reason is that the synthetic trace is a miniature version of the original program trace and does not touch all memory pages as does the real program trace—this is exactly where the simulation speedup comes from through statistical simulation—and therefore the virtual to physical address mapping is very different for the synthetic trace than for the original trace. This changes the conflict be-

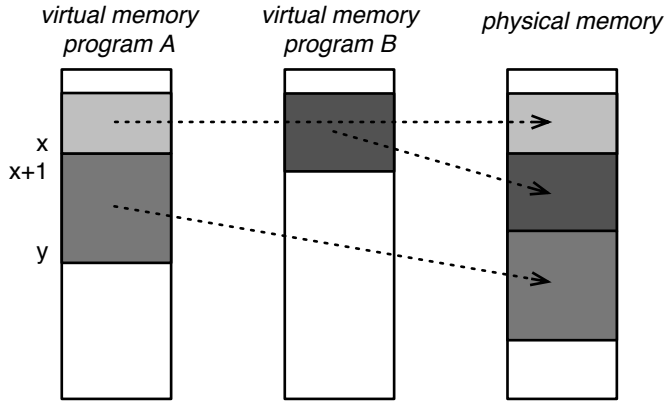

Figure 5.5: Illustrating virtual to physical address translation during statistical simulation.

havior in the memory hierarchy during statistical simulation compared to detailed simulation, yielding very different performance pictures. To solve this problem, we propose a simple but effective strategy, as illustrated in Figure 5.5. Say the last virtual memory page touched by a program is page x, and the next memory access (for the same program) touches virtual memory page y, see program A in Figure 5.5. Then, the virtual to physical address mapper will allocate virtual memory pages $x + 1$ up to y in the next available physical memory, i.e., the bump pointer is advanced by $y - x$ memory pages. This assumes that the original program accesses memory pages $x + 1$ up to $y - 1$ prior to accessing memory page y; we found this simple heuristic to be a reasonable approximation because of spatial locality—(virtual) memory pages that are next to each other are most likely to be allocated close in time.

5.1.4 Multibank DRAM modeling

When we are interested in design space exploration of more realistic DRAM with multiple banks, we could collect the microarchitecture-dependent bank/row miss probabilities per memory reference in the SFG. However, this would limit the applicability of statistical simulation. In an attempt to model the DRAM characteristics independently of the microarchitecture, we have observed that using bank reuse and row reuse information derived from the synthetic address stream, did not yield accurate results for a synthetic traces of ten million instructions (for example the absolute IPC error for equake and

facerec is 44% and 25%, respectively). However, we believe that using longer traces would solve this—synthetic addresses were used in SMART [40, 41]—but this would dramatically increase simulation time and would jeopardize the usefulness of statistical simulation. We therefore take a different approach in order to reduce the overhead of the microarchitecture-dependent DRAM modeling.

As mentioned in Section 5.1.1, we collect main memory row stack depth distributions dependent on the virtual address assuming a single-bank DRAM design. In addition, we collect the following conditional probabilities: (i) $P(\ H_b \mid B = b\ )$ represents the probability of a bank hit (H_b) given an access to bank $B = b$, and (ii) $P(\ H_r \mid (B = b \wedge D = d)\ )$ represents the probability of a row hit (H_r) given an access to bank $B = b$ and given a single-bank-DRAM row stack depth $D = d$. Both probabilities depend on the number of banks and their organization (interleaved or linear) as well as row size. However, we compute these probabilities for different DRAM organizations simultaneously in one profiling run. We also collect aggregate probabilities per profile interval, i.e., they are computed *independently* from the context of the SFG. We found that using these aggregate statistics yields good accuracy as we will show in the evaluation, see Section 5.3.

The bank hit conditional probability is computed as:

$$P(\ H_b \mid B = b\ ) = \frac{P(\ B_0 = b \wedge B_1 = b\ )}{P(\ B_1 = b\ )},$$

with B_0 and B_1 two consecutively accessed banks. The access corresponding with B_1 results in a bank hit if both B_0 and B_1 equal the same bank b.

The row hit conditional probability is computed as:

$$P(\ H_r \mid (B = b \wedge D = d)\ ) = \frac{P(\ \bigwedge_{i=1}^{d}(B_i \neq b) \wedge (B_{d+1} = b) \mid D = d\ )}{P(\ B_{d+1} = b \mid D = d\ )},$$

with $B_0 \ldots B_{d+1}$ $d + 2$ consecutively accessed banks, and d the (single-bank) DRAM row stack depth. The access corresponding with B_0 touches the same row as the access corresponding with B_{d+1} and thus $B_0 = B_{d+1} = b$. The access corresponding with B_{d+1} will result in a row hit in a multibank DRAM configuration if all the intermediately accessed banks $B_1, \ldots, B_d$ differ from b.

5.1.5 Simulating load and store instructions

As mentioned before, we generate virtual address and L1/L2/DRAM stack depths for each memory reference. Simulating the synthetic trace on a CMP then requires that we effectively simulate the entire memory hierarchy. In statistical simulation for a single-core processor system on the other hand, the memory hierarchy does not need to be simulated since cache misses are simply flagged as such in the synthetic trace; appropriate latencies are assigned based on these cache miss flags, see Chapters 3 and 4, and [19, 59, 61]. Statistical simulation of a CMP with shared memory hierarchy resources, on the other hand, requires the simulation of caches, DRAM and their interconnections in order to model conflict behavior. In the following two subsections, we will explain how we effectively simulate the caches and the main memory, respectively.

Simulating synthetic caches

Every cache line in each cache contains the following information:

- The ID of the program that most recently accessed the cache line; we will refer to this ID as the *program ID*. This enables the statistical simulator to keep track of the program *owning* the cache line.
- The set index of the set in the largest cache of interest that corresponds to the given cache line; we will refer to this set index as the *stored set index*. In case the simulated cache has as many sets as the largest cache of interest, the stored set index is the set index of the simulated cache. The stored set index will enable the statistical simulator to model cache lines conflicting for a given set in case the number of sets is reduced for the simulated cache compared to the largest cache of interest.
- A valid bit stating whether the cache line is valid.
- A cold bit stating whether the cache line has been accessed. The cold bit will be used for driving cache warm-up as will be discussed later.
- In case of a write-back cache, we also maintain a dirty bit stating whether the cache line has been written by a store operation.

- And finally, we also keep track of which instruction in the synthetic trace accessed the given cache line; this is done by storing the position of the instruction in the synthetic trace which we call the *instruction ID*.

Simulating a cache then proceeds as follows, assuming that all memory references are annotated with virtual address va, memory location reuse distance rd, and stack depth information sd for the largest cache of interest. The virtual address is translated into the physical address pa upon a memory access, as explained in Section 5.1.3. The pseudo code for a cache access is given in Figure 5.6.

Upon a cache access (line 1), we search in the MSHRs whether a delayed hit—secondary miss—occurred (lines 3–4). If not, we determine the set s being accessed in the simulated cache from the physical address (line 6)—assuming a physically indexed cache. This is done by selecting the appropriate $\log_2(S)$ bits from the address pa, with S being the number of sets in the simulated cache. Moreover, we compute the largest cache set index s'—stored set index—from pa by selecting the appropriate $\log_2(S')$ bits (line 7), with S' being the number of sets in the original largest cache considered during the profiling step. Thereafter, we walk over all cache lines in set s to determine whether a hit or miss occurred (lines 10–26).

The cache access is a cache hit (lines 20–24) in case there are more than sd valid cache lines in set s (line 12) for which (i) the stored set indices equal s' (line 13) and (ii) the stored program IDs equal the ID of the program being simulated (line 14). Upon a cache hit, we update the LRU state in set s (line 21). In case the above conditions do not hold, the cache access is considered a cache miss, and we allocate a free MSHR (lines 28–29)—we update the LRU state when the miss is resolved.

An appropriate warm-up approach is required for the large caches, such as the unified L2 caches. Without appropriate warm-up, the large caches would suffer from a large number of cold misses. Making the synthetic trace longer could solve this problem, but this would definitely affect the usefulness of statistical simulation which is to provide performance estimates from very fast simulation runs. For this reason we take a different approach and use a warm-up strategy for warming the L2 cache. The warm-up technique that we use first initializes all cache lines as being cold by setting the cold bit in all cache lines. The warm-up approach then applies a *hit-on-cold* strategy [73], i.e., upon the first access to a given cache line we assume it is a hit and the cold bit

```
 1:  access( prog_ID, insn_ID, phys_addr, cache_line_reuse_distance, cache_line_stack_depth) {
 2:  |
 3:  |   if ( find_MSHR( prog_ID, insn_ID, phys_addr, cache_line_reuse_distance)
 4:  |      return secondary_miss;
 5:  |
 6:  |   s  = extract_set(phys_addr);
 7:  |   s' = extract_stored_set_idx(phys_addr);
 8:  |   cache_line_count = 0;
 9:  |
10:  |   for (all cache lines in set s) {
11:  |   |
12:  |   |   if ( ( cache_line.status == Valid &&
13:  |   |   |      cache_line.stored_set_idx == s' &&
14:  |   |   |      cache_line.prog_ID == prog_ID
15:  |   |   |    )
16:  |   |   |    || cache_line.status == Cold ) {
17:  |   |   |
18:  |   |   |   cache_line_count++;
19:  |   |   |
20:  |   |   |   if (cache_line_count > cache_line_stack_depth) {
21:  |   |   |       update LRU state;
22:  |   |   |       cache_line.reset_cold_bit();
23:  |   |   |       return hit;
24:  |   |   |   }
25:  |   |   }
26:  |   }
27:  |
28:  |   allocate new MSHR;
29:  |   return miss;
30:  }
```

Figure 5.6: High-level pseudocode for simulating caches with LRU replacement policy during synthetic trace simulation.

is set to zero (line 22). In other words, if the cold bit is set (line 16), we assume it is a valid cache line for which the stored set index equals s' and the stored program ID equals the ID of the program issuing the access. This hit-on-cold warm-up strategy is simple to implement, and is fairly accurate [73].

During this work, we also found that it is important to model L1 D-cache write-backs during synthetic trace simulation; write-backs can have a significant impact on the conflict behavior in the shared L2 cache. This is done by simulating the L1 D-cache similarly to what is described above; L1 D-cache write-backs then access the L2 cache. We use a simple heuristic to determine the outcome (in terms of hit/miss) of a write-back accessing the L2 cache. The L2 cache access is assumed to be a miss if all instruction IDs in the given set (with the same program ID) are larger than the instruction ID of the cache line written in the L2. In other words, a miss is assumed if all of these instructions are executed later than the last store to the cache line, which causes a write-back when the (dirty) cache line is evicted. Otherwise, it is assumed to be a hit.

```
 1:  access( models, prog_ID, phys_addr, row_stack_depth) {
 2:  |
 3:  |   b = extract_bank(phys_addr);
 4:  |
 5:  |   if (prog_ID == active_bank_prog_ID[b]) {
 6:  |      bank_hit = (rand < models.prob_bank_hit( num_banks, b)) ? 1 : 0;
 7:  |      row_hit = (rand < models.prob_row_hit( num_banks, b, row_stack_depth)) ? 1 : 0;
 8:  |   }
 9:  |   else {
10:  |      bank_hit = (b == active_bank) ? 1 : 0;
11:  |      row_hit = 0;
12:  |   }
13:  |
14:  |   return access_latency( bank_hit, row_hit);
15:  }
```

Figure 5.7: High-level pseudocode for simulating DRAM during synthetic trace simulation.

Simulating synthetic DRAM

As mentioned in Section 5.1.4, the synthetic trace also contains DRAM probability models for all main memory organizations of interest. These models are used when a synthetic instruction accesses main memory (see line 1 in the pseudo-code of Figure 5.7). A memory request in synthetic trace simulation is annotated with a physical address pa and DRAM row stack depth information sd; note that the latter expresses the temporal locality of a memory location in the address stream assuming a single-bank DRAM design. When the request accesses main memory, we first determine the bank address b from the physical address pa (line 3).

In case the program ID of the thread doing the access equals the program ID of the thread that *owns* the active bank (lines 5–8) we look up the probability of a bank hit for the given bank b in the DRAM model—$P(\ H_b \mid B = b\ )$—and use it with a random number in the interval $[0, 1]$ to determine whether a bank hit/miss occurred (line 6). (We do this the same way as described in Figure 3.3: we use a random number with the inverse cumulative distribution function to determine the particular value for a program characteristic when generating the synthetic trace.) Similarly, we determine a row hit/miss (line 7) using a random number with the probability of a row hit given the bank b and the row stack depth sd—$P(\ H_r \mid (B = b \wedge D = sd)\ )$. On the other hand, in case the active bank is *owned* by another thread (lines 9–12) the access results in a bank hit if the active bank equals b (line 10), and it always results in a row miss (line 11)—virtual memory pages from different co-executing programs are mapped to different rows. Finally, we return the access latency according to the bank/row hit/miss outcome (line 14).

	recompute profile	
microarchitectural parameter	*single-core (Chapter 4)*	*multicore (Chapter 5)*
number of cache levels	yes	yes
cache size	yes	no
line size	yes	yes
associativity	yes	no
replacement policy	yes	yes
line updating policy	yes	no
access latencies	no	no
off-chip memory bandwidth	no	no
number of DRAM banks	yes	yes (in DRAM profile)
interleaved vs. linear memory access control	yes	yes (in DRAM profile)
DRAM row size	yes	yes
DRAM access latencies	no	no

Table 5.2: Comparing which memory hierarchy parameters that do or do not require that a new statistical profile is computed for single-core statistical simulation versus multicore statistical simulation.

5.1.6 Design space exploration using statistical simulation

In Chapter 3 we have shown that a statistical profile contains both microarchitecture-dependent and microarchitecture-independent characteristics. This has an important implication in practice: the use of a single statistical profile is limited by the set of microarchitecture-dependent characteristics. In this chapter we proposed new cache statistics to model the conflict behavior in shared caches. An additional advantage over single-core statistical simulation, as described in Chapters 3 and 4, is that the cache statistics are largely microarchitecture-independent. In such way, we can explore most of the memory hierarchy design space from a single statistical profile. The only parameter that requires a new statistical profile to be computed is the number of cache levels and their line sizes. The number of cache sets, cache associativity, bandwidth and latencies can be changed without recollection of the statistical profile. Table 5.2 summarizes the difference in microarchitecture-dependent memory statistics between single-core and multicore statistical simulation. The DRAM statistics on the other hand are microarchitecture-dependent. However, we decoupled these

statistics from the SFG. Furthermore, we can collect the DRAM bank access sequence probabilities for various kinds of DRAM configurations simultaneously in one profiling run. In such way, we can still explore a DRAM design space with only one SFG profile. When simulating, we only need to use the proper DRAM bank access sequence probabilities corresponding with the DRAM configuration.

5.2 Experimental setup

We use the SPEC CPU2000 benchmarks with the reference inputs in our experimental setup, see Table 5.3; this table also displays the global L2 cache miss rates for the various benchmarks in our baseline 16 MB 16-way set-associative cache. The binary programs of the CPU2000 benchmarks are taken from the SimpleScalar website. We consider one-hundred-million-instruction single (and early) simulation points as determined by SimPoint [64, 67] in all of our experiments. The synthetic traces are ten million instructions long, unless mentioned otherwise—we evaluate the impact of the synthetic trace length on accuracy and simulation speedup in Section 5.3.2. For measuring the statistical profiles capturing time-varying behavior, we measure a statistical profile per interval of ten million instructions. From these ten statistical profiles, we then generate ten mini-traces of one million instructions each. Subsequently, we coalesce these mini-traces to form a synthetic trace of ten million instructions.

We use the M5 simulator [6] in all of our experiments. Our baseline per-core microarchitecture is a 4-wide superscalar out-of-order core, see Table 5.4. When simulating a CMP, we assume that all cores share the L2 cache as well as the off-chip bandwidth for accessing main memory. Simulation stops as soon as one of the co-executing programs terminates, i.e., as soon as one of the programs has executed one hundred million instructions in case of detailed simulation, or ten million instructions in case of statistical simulation. We then record how many instructions were executed so far for each co-executing program, and we compute single-threaded IPC for executing that many instructions.

Having obtained IPC numbers under both multicore execution and single-threaded execution, we can compute system throughput (STP) proposed by Eyerman and Eeckhout [25], also called weighted speedup [68], and average normalized job turnaround time (ANTT), which is the reciprocal of the harmonic mean proposed by Luo et

benchmark	input	simpoint	L2 miss rate
bzip2	program	9	1.4%
crafty	ref	0	0.7%
eon	rushmeier	18	0.0%
gap	ref	2,094	0.7%
gcc	166	99	0.8%
gzip	graphic	9	2.3%
mcf	ref	316	24.8%
parser	ref	16	2.3%
perlbmk	makerand	1	0.2%
twolf	ref	31	5.3%
vortex	ref2	57	0.5%
vpr	route	71	2.9%
ammp	ref	2,130	4.8%
applu	ref	18	4.6%
apsi	ref	46	2.5%
art	ref-110	67	32.8%
equake	ref	194	8.3%
facerec	ref	136	2.7%
fma3d	ref	298	2.2%
galgel	ref	3,150	3.2%
lucas	ref	35	8.9%
mesa	ref	89	0.1%
mgrid	ref	6	3.3%
sixtrack	ref	82	0.2%
swim	ref	5	7.5%
wupwise	ref	584	1.3%

Table 5.3: The SPEC CPU2000 benchmarks, their reference inputs, the single one hundred million simulation points used in this dissertation and their global miss rates for a 16 MB 16-way set-associative L2 cache.

Processor core	
ROB	80 entries
LSQ	64 entries
store buffer	32 entries
processor width	decode, dispatch, issue and commit 4 wide fetch 8 wide
latencies	load (2), mul (3), div (20)
L1 I-cache	32 KB 4-way set-assoc 64 B line size
L1 D-cache	32 KB 4-way set-assoc 64 B line size, 8 MSHRs, 16 MSHR targets, 16-entry write buffer
branch predictor	10 Kbit local predictor, 8-way set-assoc 2 K-entry BTB, 32-entry RAS
Shared unified L2 cache	
cache size	16 MB
cache associativity	16-way set-assoc
line size	64 B
access latency	12 CPU cycles
MSHRs	8
MSHR targets	16
write buffer	16 entries
Bus	
bus frequency	666 MHz
bus width	16 B
SDRAM	
banks	4
row size	4 KB
RAS active time	125 CPU cycles
RAS to CAS delay	25 CPU cycles
CAS latency	5 CPU cycles
RAS precharge time	10 CPU cycles
RAS cycle time	70 CPU cycles

Table 5.4: Baseline processor core model assumed in our experimental setup; simulated CMP architectures share the L2 cache.

al. [52]. System throughput is a system-oriented performance metric, and is defined as:

$$STP = \sum_{i=1}^{n} \frac{IPC_{i,multicore}}{IPC_{i,single\_threaded}},$$

with n the number of co-executing programs—this is a bigger-is-better metric. Average normalized job turnaround time is a user-oriented performance metric, and is defined as:

$$ANTT = \frac{1}{n} \sum_{i=1}^{n} \frac{IPC_{i,single\_threaded}}{IPC_{i,multicore}},$$

Ideally the turnaround time is not affected and $ANTT = 1$— bigger than one is worse. We will report both metrics in experiments in the evaluation section where it is appropriate .

5.3 Evaluation

We now evaluate the statistical simulation technique proposed in this chapter along three dimensions: (i) accuracy for a single design point as well as for a wide design space, (ii) simulation speed, and (iii) storage requirements for storing the statistical profiles on disk.

5.3.1 Accuracy

Homogeneous workloads

The top graph in Figure 5.8 evaluates the accuracy of statistical simulation for a single program running on a single-core processor. The average IPC prediction error is 2.4%; this is in line with previously reported results in Chapter 4. The other three graphs in Figure 5.8 evaluate the accuracy when running homogeneous multiprogram workloads, i.e., multiple copies of the same program are executed simultaneously on different cores. The average IPC prediction errors for the two-core, four-core and eight-core machines are 5.6%, 6.3% and 7.3%, respectively—the increasing error for a higher number of cores can be intuitively understood because of the increasing contention for the shared resources.

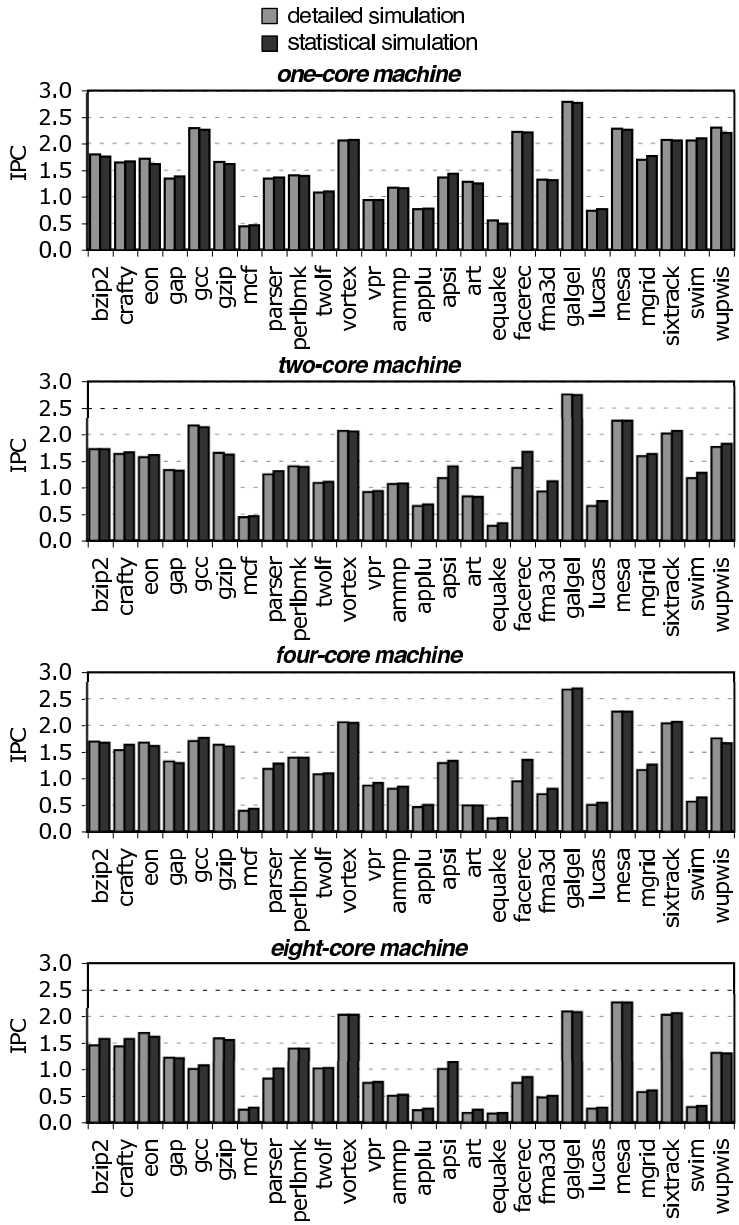

Figure 5.8: Evaluating the accuracy of statistical simulation for single-program and homogeneous multiprogram workloads.

Statistical simulation is capable of accurately tracking the impact of the shared L2 cache on overall application performance. For some programs, cache sharing has almost no impact, see for example mesa: the IPC for mesa remains unaffected by L2 cache sharing as we increase the number of co-executing copies. For other programs on the other hand, cache sharing has a large impact, see for example art, mgrid and swim, for which performance degrades as more copies co-execute. Statistical simulation is accurate enough for identifying which programs are susceptible to L2 cache sharing; moreover, statistical simulation yields an accurate prediction of the extent to which cache sharing affects overall performance.

Heterogeneous workloads

Figure 5.9 evaluates the accuracy of statistical simulation for heterogeneous workloads. The four sets of graphs, labeled (a), (b), (c) and (d) represent different sets of workloads. The left column shows results through detailed simulation; the right column shows results through statistical simulation. In each graph, there are four bars for each benchmark. The *one-core* bars represent per-benchmark IPC when run alone. The *two-core* bars represent per-benchmark IPC when co-run with another benchmark; the *four-core* bars represent per-benchmark IPC when co-run with three other benchmarks, etc. The co-run workloads are determined as such, see for example the top-left graph: we co-run art with applu, and mcf with lucas, etc. on a two-core configuration; for the four-core configuration, we co-run art, applu, mcf and lucas, and we co-run equake, wupwise, swim and facerec; for the eight-core configuration we co-run all benchmarks in the workload.

Not surprisingly, per-benchmark IPC decreases with increasing multicore processing. This is due to resource conflicts in the shared memory hierarchy, i.e., the more conflicts, the more the co-executing programs interact and affect each other's performance. The degree to which co-executing benchmarks affect each other's performance heavily depends on the benchmarks' characteristics, i.e., the more memory-intensive the benchmarks are, the more they affect each other's performance. For example, the co-executing benchmarks affect each other's performance very heavily in the workload (a)—these benchmarks are all memory-intensive; on the contrary, the benchmarks in workload (c) barely affect each other's performance because none of the benchmarks are memory-intensive. The important observation from these graphs

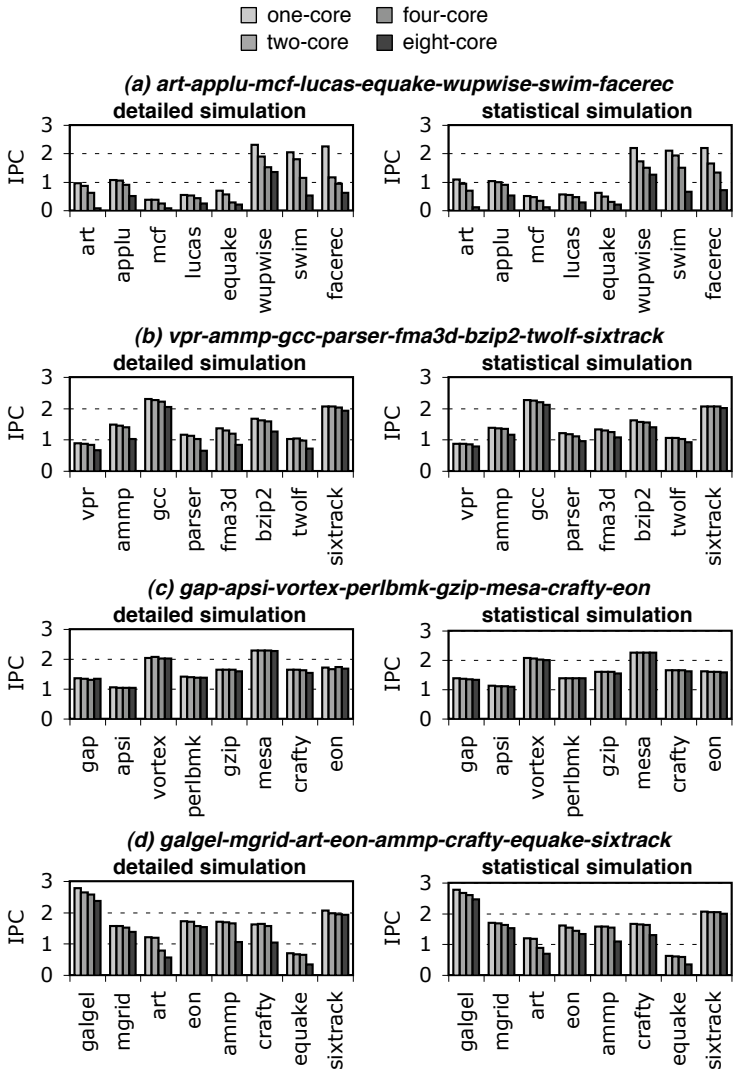

Figure 5.9: Evaluating the per-core accuracy of statistical simulation for four heterogeneous workload mixes. The two-core configuration results show per-benchmark IPC when co-run with another benchmark; e.g., in the top-left graph, we co-run art with applu, mcf with lucas, etc. For the four-core configuration, we co-run art, applu, mcf and lucas, and we co-run equake, wupwise, swim and facerec; for the eight-core configuration we co-run all benchmarks in the workload.

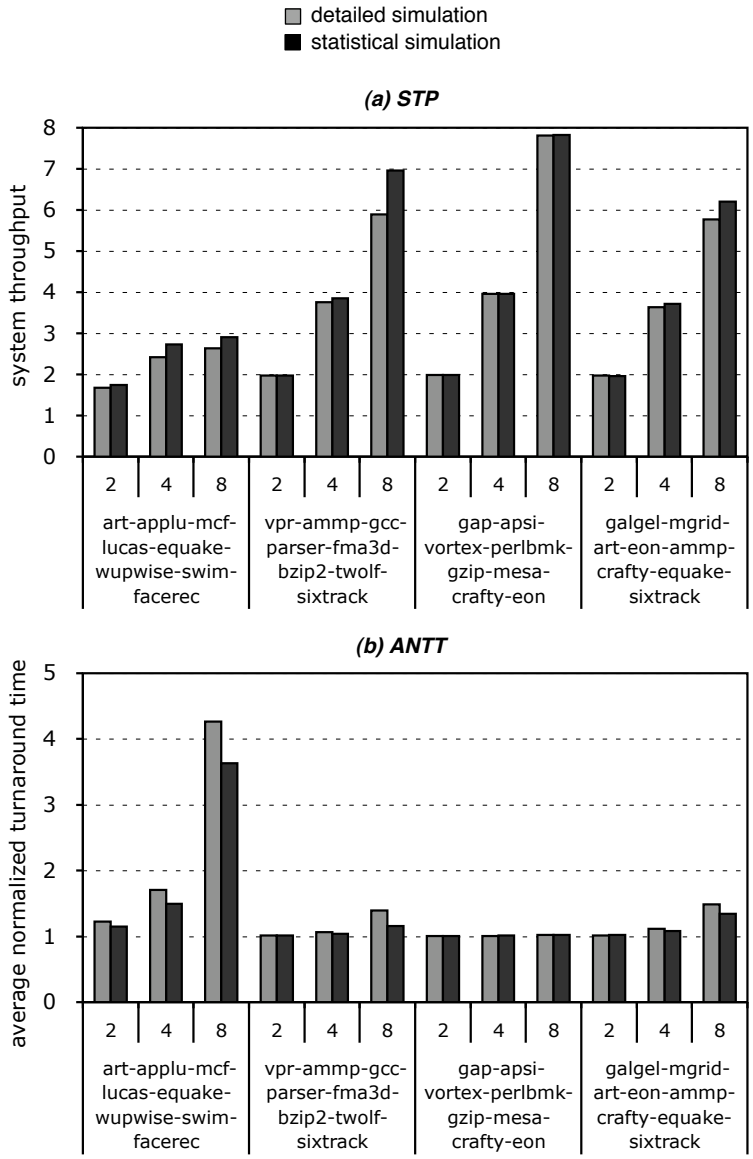

Figure 5.10: Evaluating the accuracy of statistical simulation for heterogeneous workload mixes in terms of STP (a) and ANTT (b).

is that statistical simulation accurately tracks the performance trends observed through detailed simulation. Figure 5.10 shows the system throughput (a) and the average normalized job turnaround time (b); the average STP error and ANTT error equals 4.9% and 5.6%, respectively.

Design space exploration

We now demonstrate the accuracy of statistical simulation for driving design space exploration, which is the ultimate goal of the statistical simulation technique. To do so, we consider a design space of 60 design points with varying L2 cache configurations and a varying number of cores. We vary the L2 cache size from 128 KB to 16 MB with varying associativity from 2- to 16-way set-associative; the cache line size is kept constant at 64 bytes. And we vary the number of cores from 2, 4 up to 8. This design space consisting of 60 design points is very small compared to a realistic design space, however, the reason is that we are validating the accuracy of statistical simulation against detailed simulation. The detailed simulation for all those 60 design points was very time-consuming, which is the motivation for statistical simulation in the first place.

Figures 5.11 and 5.12 show a scatter plot with system throughput through detailed simulation on the horizontal axis versus system throughput through statistical simulation on the vertical axis. The four graphs in Figure 5.11 and 5.12 show four different heterogeneous eight-program mixes. The average system throughput prediction error equals 3.5%. We observe that the prediction error increases slightly with an increasing number of cores: an average prediction error of 1.9% for a two-core processor, 3.7% for a four-core processor, and 5% for an eight-core processor. Overall, we conclude that for all four workload mixes, the system throughput estimates through statistical simulation correlate very closely with the system throughput numbers obtained from detailed simulation.

Cache design space exploration

Figure 5.13 illustrates the accuracy of statistical simulation for exploring the shared L2 cache design space. In these graphs, we consider the IPC for the benchmarks, twolf and ammp, that we found to be sensitive to both the cache configuration parameters and the amount of parallel processing. In these experiments, we co-execute multiple copies of

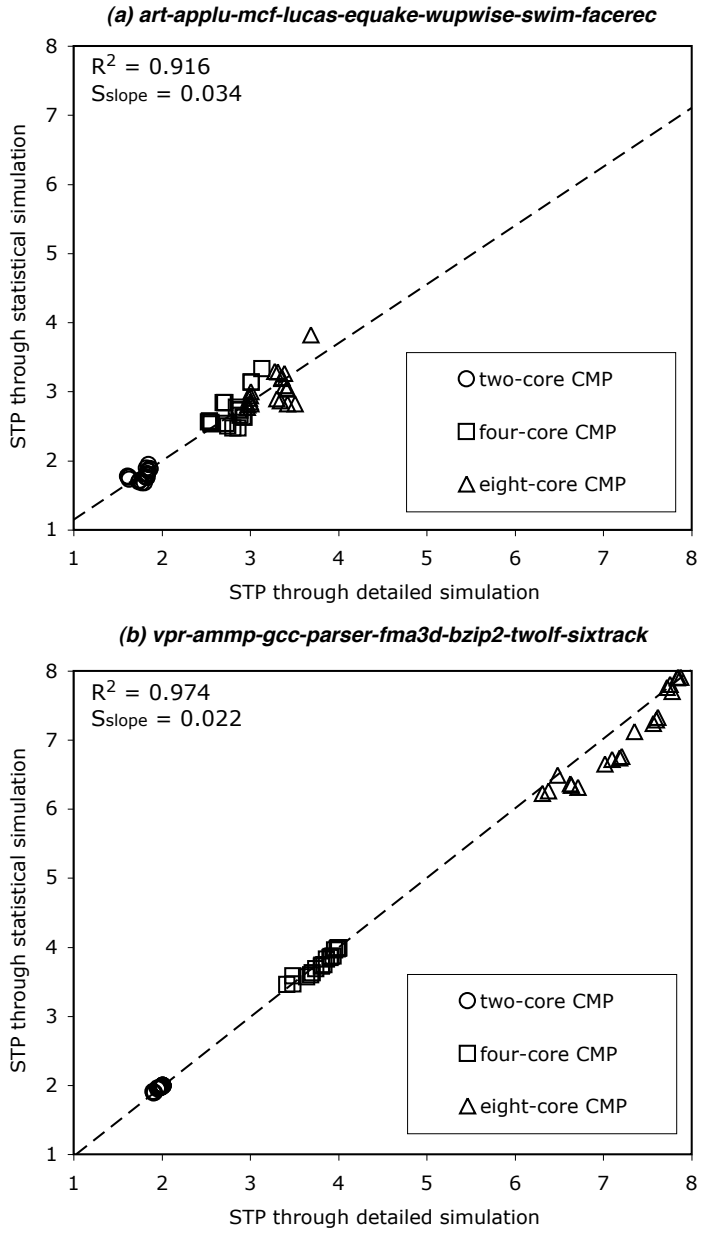

Figure 5.11: Evaluating the accuracy of statistical simulation for exploring CMP design spaces: measured system throughput through detailed simulation versus estimated system throughput through statistical simulation.

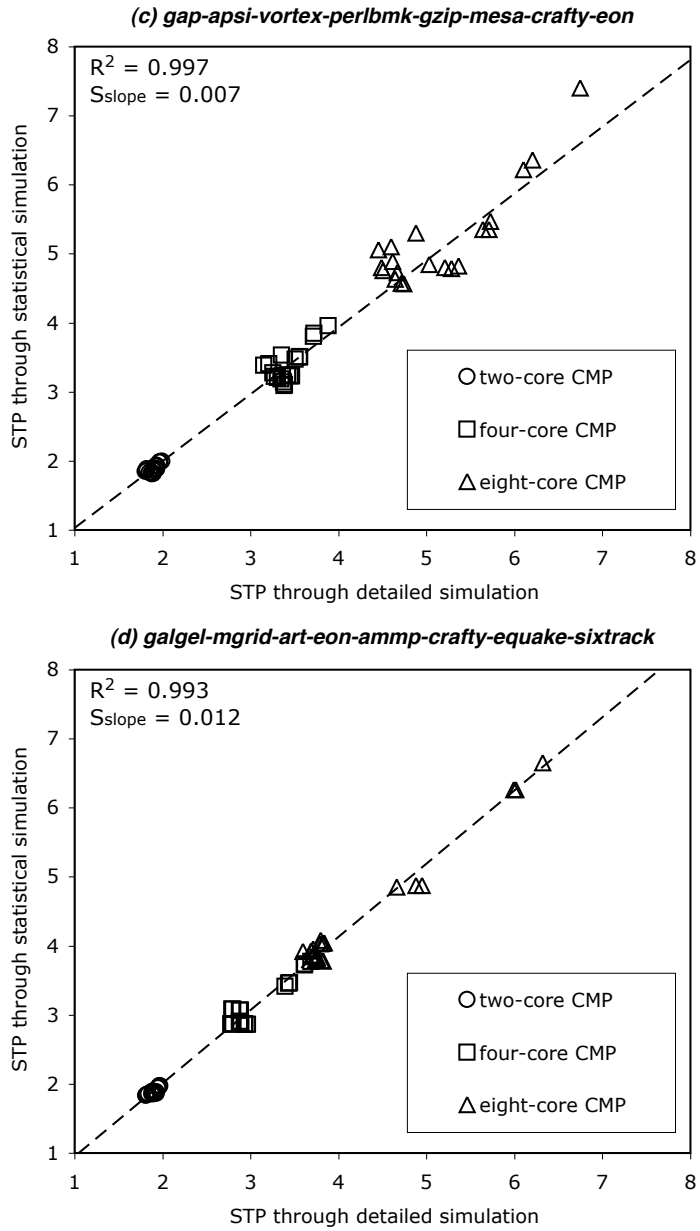

Figure 5.12: Evaluating the accuracy of statistical simulation for exploring CMP design spaces: measured system throughput through detailed simulation versus estimated system throughput through statistical simulation.

the same program on a multicore processor. Again, the overall conclusion is that statistical simulation accurately tracks performance differences across cache configurations and across a different number of cores. Note that these results were obtained from a single statistical profile, namely a statistical profile for the largest cache of interest, a 16 MB 16-way set-associative cache. In other words, a single statistical profile is sufficient to drive a cache design space exploration.

DRAM design space exploration

In this experiment we validate the multibank DRAM modeling of Section 5.1.4. Figure 5.14 shows the accuracy of statistical simulation when varying the DRAM organization in terms of the number of banks and their organization (interleaved or linear). In these graphs, we consider the STP for the four heterogeneous eight-program mixes. We observe that statistical simulation accurately tracks performance differences across these DRAM configurations. These results were obtained from a single statistical profile that assumes a single-bank design.

3D stacking case study

For demonstrating the value of statistical simulation for exploring new architecture paradigms, we now consider a case study in which we evaluate performance of a multicore processor in combination with 3D stacking [48]. In this case study, we compare the performance of a 4-core processor with a 16 MB L2 cache connected to external DRAM memory through a 16-byte wide memory bus against an 8-core processor with integrated on-chip DRAM memory (through 3D stacking) and no L2 cache and a 128-byte wide memory bus. We assume a 150-cycle access time for external memory and a 125-cycle access time for 3D-stacked memory. Figure 5.15 quantifies system throughput for these two design points for four eight-benchmark mixes. The 8-core processor with 3D stacked memory achieves substantially higher system throughput than the 4-core processor with the on-chip L2 cache. This increase in system throughput is offset by a decrease in job turnaround time. The improvement in system throughput and the reduction in turnaround time varies across workload mixes, and statistical simulation accurately tracks performance differences between both design alternatives: the maximum error in predicting the system throughput

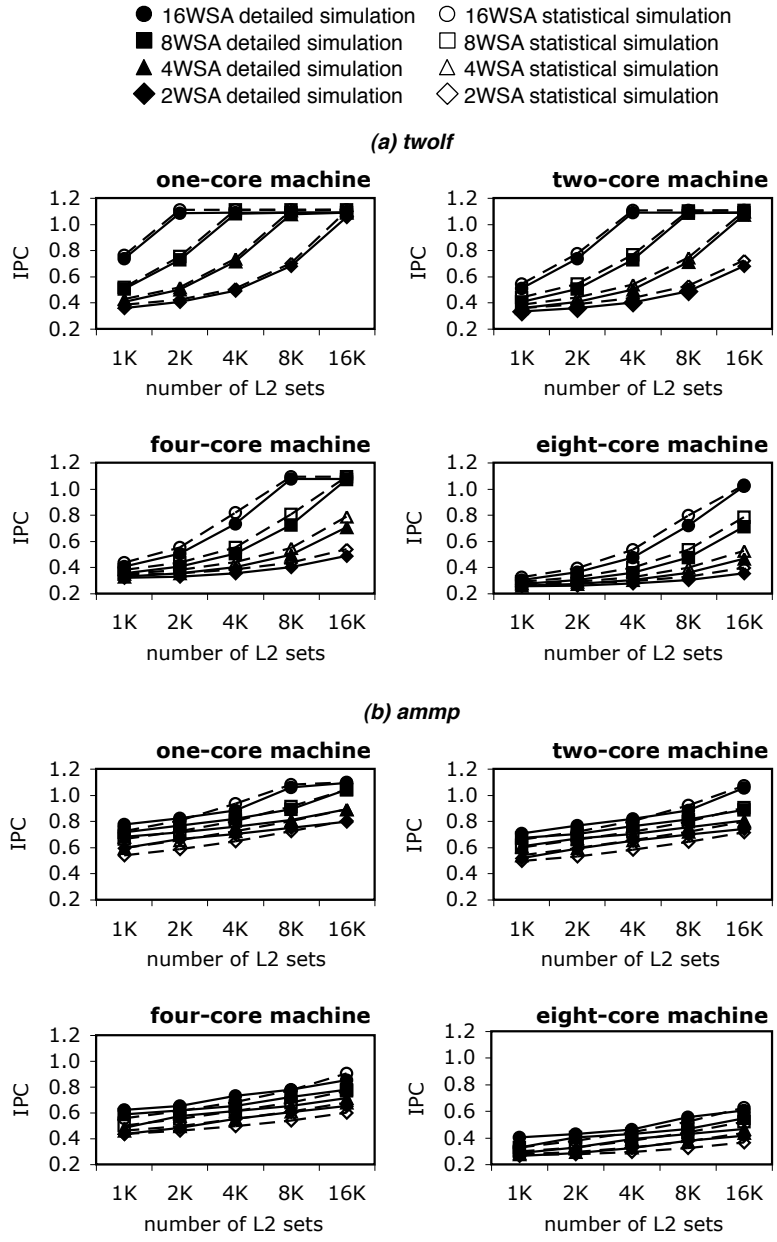

Figure 5.13: Evaluating the accuracy of statistical simulation for tracking shared cache performance as a function of the cache configuration (number of sets and associativity) and the number of cores on the CMPs; the example benchmarks are twolf and ammp.

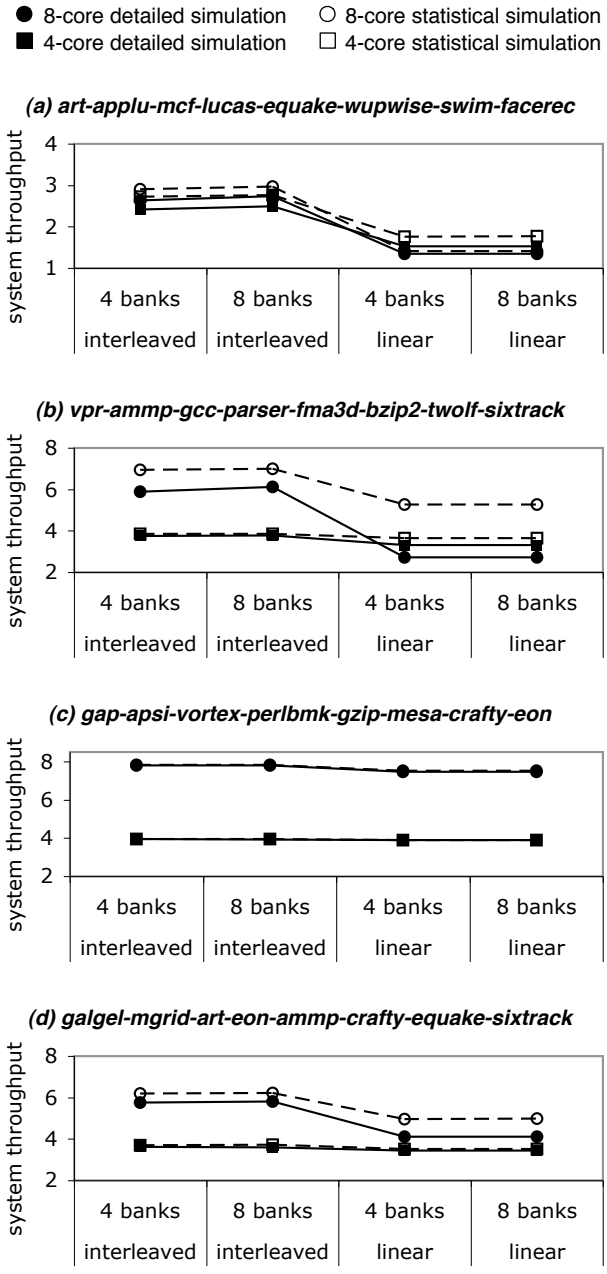

Figure 5.14: Evaluating the accuracy of statistical simulation in terms of system throughput for various DRAM configurations.

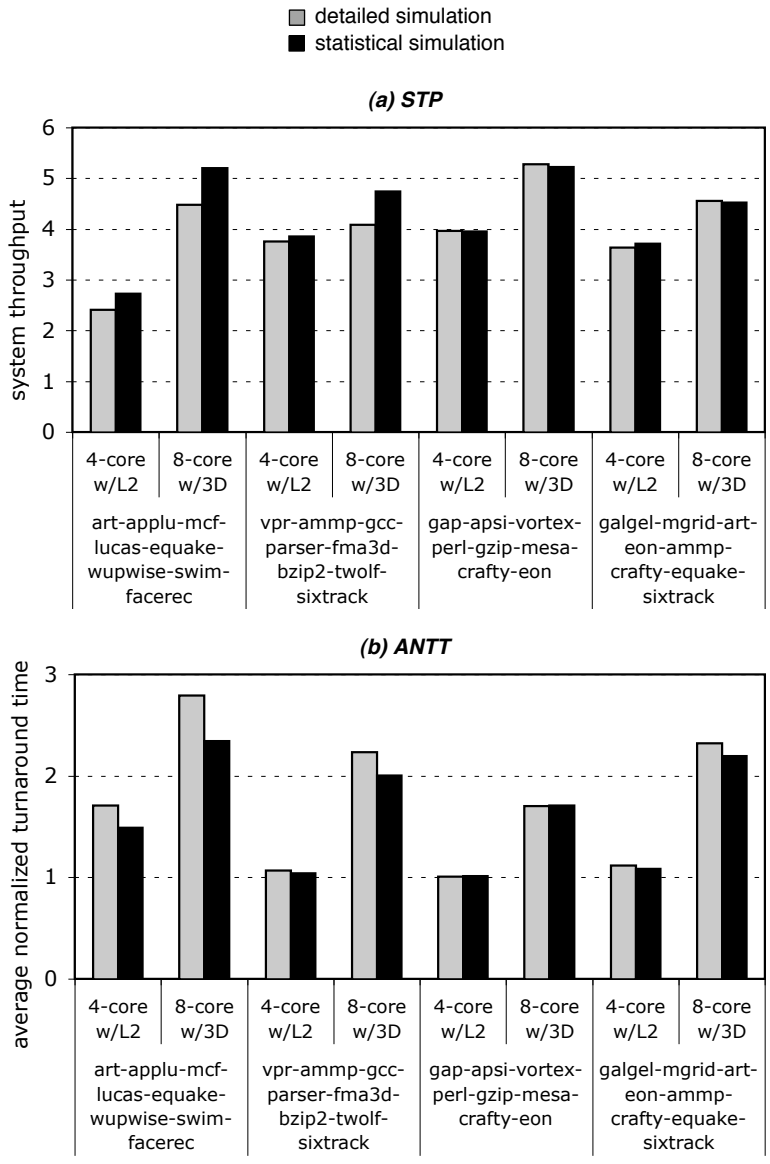

Figure 5.15: 3D stacking case study: comparing system throughput for a 4-core CMP with L2 cache and external DRAM memory versus an 8-core CMP with on-chip DRAM memory (through 3D stacking) and without an L2 cache.

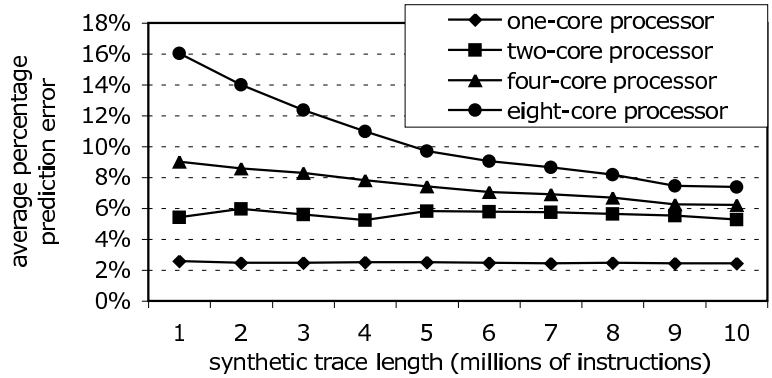

Figure 5.16: Average IPC prediction error as a function of synthetic trace length for single-program and multiprogram homogeneous workloads.

delta between the 4-core with on-chip L2 versus the 8-core with 3D-stacked DRAM is 12%.

5.3.2 Simulation speed

Having shown the accuracy of statistical simulation for CMP design space exploration, we now evaluate the simulation speed. Figure 5.16 shows the average IPC prediction error as a function of the synthetic trace length. For a single-program workload, the prediction error curve stays almost flat, i.e., increasing the size of the synthetic trace beyond one million instructions does not increase prediction accuracy. For multiprogram workloads on the other hand, the prediction accuracy is sensitive to the synthetic trace length, and sensitivity increases with the number of programs in the multiprogram workload. This can be understood intuitively: the more programs there are in the multiprogram workloads, the longer it takes before the shared caches are warmed up and the longer it takes before the conflict behavior is appropriately modeled between the co-executing programs. The results in Figure 5.16 demonstrate that ten million instruction synthetic traces yield accurate performance predictions, even for eight-core processors. In our experiments we therefore went from one hundred million instruction real program traces to ten million instruction synthetic traces. This is a factor ten decrease in the dynamic instruction count which yields an approximate factor ten reduction in the overall simulation time.

5.3.3 Storage requirements

As a final note, the storage requirements are modest for multicore statistical simulation. The statistical profiles when compressed on disk are 87 MB on average per benchmark. However, the (compressed) profile sizes of single-core statistical simulation are on average twenty times smaller than those of multicore statistical simulation, see Chapter 4. This increase is mainly due to the virtual address distributions that are part of the statistical profile.

5.4 Summary

In this chapter we have enhanced statistical simulation as a fast simulation technique for chip-multiprocessors running multiprogram workloads. In order to do so, we extended the statistical simulation paradigm in two ways: (i) we collect cache set access and per-set stack depth statistics and (ii) we model time-varying behavior in the synthetic traces. These two enhancements enable the accurate modeling of conflict behavior observed in shared caches. Furthermore, we have extended statistical simulation to model more realistic DRAM memory organizations.

Our experimental results using the SPEC CPU2000 benchmarks show that statistical simulation is accurate with average IPC prediction errors of less than 7.3% over a broad range of CMP design points, while being one order of magnitude faster than detailed simulation. Statistical simulation is capable of tracking performance trends across a CMP design space, i.e., in our experiments we vary the number of cores, the cache configuration, and the DRAM configuration. This makes statistical simulation a viable fast simulation approach to CMP design space exploration.

Furthermore, the statistical profile developed in this chapter is less dependent on the microarchitecture. In the previous chapter we had to recompute the statistical profile when the cache configuration changed, e.g., cache size and associativity. With multicore statistical simulation the same profile can be used to model any cache configuration smaller than the (largest) cache configuration used during the profiling step. This improves the applicability of statistical simulation, i.e., a statistical profile covers a much larger design space compared to single-core statistical simulation.

A possible disadvantage of multicore statistical simulation is that accurately modeling conflict behavior in shared resources requires the synthetic traces to be longer. However, we showed that ten million instructions are sufficient for obtaining converged performance estimates.

Chapter 6

Interval Simulation

The obscure we see eventually.
The completely obvious, it seems, takes longer.
Edward R. Murrow

In this chapter we present a new simulation paradigm, i.e., interval simulation. Interval simulation reduces both simulation time and simulator development complexity by raising the level of abstraction in the core-level models. It determines the progress of the running threads through analytical interval-based models instead of through tracking the propagation of individual instructions through the pipeline stages. The analytical timing models for the individual cores consult branch predictor simulators and memory hierarchy simulators to derive the miss events and their latencies.

As mentioned in the previous chapter, there is a tight performance entanglement between co-executing threads on multicore processors. The cooperation between the mechanistic analytical model and the miss event simulators enables the modeling of this complex performance entanglement.

6.1 Interval analysis

Interval simulation builds on a recently developed mechanistic analytical performance model, i.e., interval analysis [26], which we briefly revisit here.

With interval analysis, execution time is partitioned into discrete

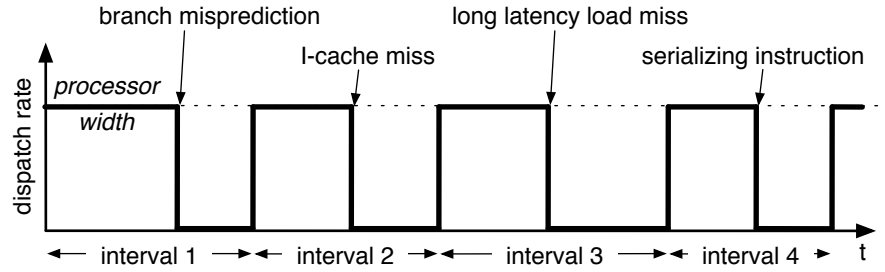

Figure 6.1: Interval analysis analyzes performance on an interval basis determined by disruptive miss events.

intervals by disruptive miss events such as cache misses, TLB misses, branch mispredictions and serializing instructions. The basis for the model is that an out-of-order processor is designed to smoothly stream instructions through its various pipelines and functional units. Under optimal conditions (no miss events), a well-balanced design sustains a level of performance more-or-less equal to its pipeline front-end *dispatch* width—we refer to dispatch as the point of entering the instructions from the front-end pipeline into the reorder buffer and issue queues.

The interval behavior is illustrated in Figure 6.1 which shows the number of dispatched instructions on the vertical axis versus time on the horizontal axis. By dividing execution time into intervals, one can analyze the performance behavior of the individual intervals. In particular, one can describe and determine the performance penalty per miss event based on the type of interval (the miss event that terminates it):

- For an *I-cache miss (or I-TLB miss)*, the penalty equals the miss delay, i.e., the time to access the next level in the memory hierarchy.
- For a *branch misprediction*, the penalty equals the time between the mispredicted branch being dispatched and new instructions along the correct control flow path being dispatched. This penalty includes the branch resolution time plus the front-end pipeline depth.
- Upon a *long-latency load miss*, i.e., a last-level L2 D-cache load miss or a D-TLB load miss, the processor back-end will stall because of the reorder buffer (ROB), issue queue, or rename registers getting exhausted. As a result, dispatch will stall. When the miss

returns from memory, instructions at the ROB head will be committed, and new instructions will enter the ROB. The penalty for a long-latency D-cache miss thus equals the time between dispatch stalling upon a full ROB and the miss returning from memory. This penalty can be approximated by the memory access latency. In case multiple independent long-latency load misses make it into the ROB simultaneously, both will overlap their execution, thereby exposing memory-level parallelism (MLP) [16], provided that a sufficient number of outstanding long-latency loads are supported by the hardware. The penalty of multiple overlapping long-latency loads thus equals the penalty for an isolated long-latency load. In case of dependent long-latency loads, their penalties serialize.

- Chains of dependent instructions, possibly with L1 data cache misses and long-latency functional unit instructions (divide, multiply, etc.), may cause a resource (e.g., reorder buffer, issue queue, physical register file, write buffer, etc.) to fill up, resulting in a *resource stall*. Consequently, dispatch may (eventually) be stalled. The penalty or the number of cycles where dispatch stalls due to a resource stall are attributed to the instruction at the ROB head, i.e., the instruction blocking commit and thereby stalling dispatch.

- Before a *serializing* instruction enters the ROB, the processor will stall dispatch until the ROB is empty. The penalty thus equals the time needed to drain the ROB.

Interval analysis also provides good insight in how miss events overlap with each other. For example, the penalty due to an I-cache miss that follows a long-latency load miss is hidden underneath the long-latency load penalty. Similarly, the penalty for a mispredicted branch that follows a long-latency load in the dynamic instruction stream on which it does not depend, is completely hidden underneath the long-latency load. If on the other hand, the mispredicted branch depends on the long-latency load, then both penalties serialize.

6.2 Multicore interval simulation

In the following sections we will discuss the interval simulation paradigm for chip-multiprocessors[1]. First, we will clarify the basic idea and explain the general framework. Second, we will elaborate on the algorithm, which models the streaming of the instructions through the pipeline at a high level of abstraction. Finally, we will discuss the limitations of interval simulation.

6.2.1 Framework overview

The multicore interval simulation paradigm that we present in this chapter is drawn schematically in Figure 6.2. A functional simulator supplies instructions to the interval simulator which uses interval analysis for driving the timing of the individual cores. The miss events are handled by branch predictor and memory hierarchy simulators. The branch predictor simulator models the branch predictors in the individual cores and is invoked by the mechanistic multicore simulator upon the execution of a branch instruction. The branch predictor simulator returns whether or not a branch is correctly predicted. The memory hierarchy simulator models the entire memory hierarchy. This includes cache coherence, private (per-core) caches and TLBs as well as the shared last-level caches, interconnection network, off-chip bandwidth and main memory. The memory hierarchy simulator is invoked for each I-cache/TLB or D-cache/TLB access by the mechanistic multicore simulator, and returns the (miss) latency.

The interval simulator models the timing for the individual cores. The simulator maintains a *pre-execution window* of instructions for each simulated core, see Figure 6.2. This window of instructions corresponds to the reorder buffer of a superscalar out-of-order processor, and is used to determine miss events that are overlapped by long-latency load misses. The functional simulator feeds instructions into the pre-execution window.

Core-level progress (i.e., timing simulation) is derived by considering the instruction at the pre-execution window head, see also Figure 6.2. In case of an I-cache miss, we increase the core simulated time by the miss latency. In case of a branch misprediction, we increase the

[1] We believe that interval simulation can deal with shared-memory multiprocessors as well, but we do not evaluate this in this dissertation.

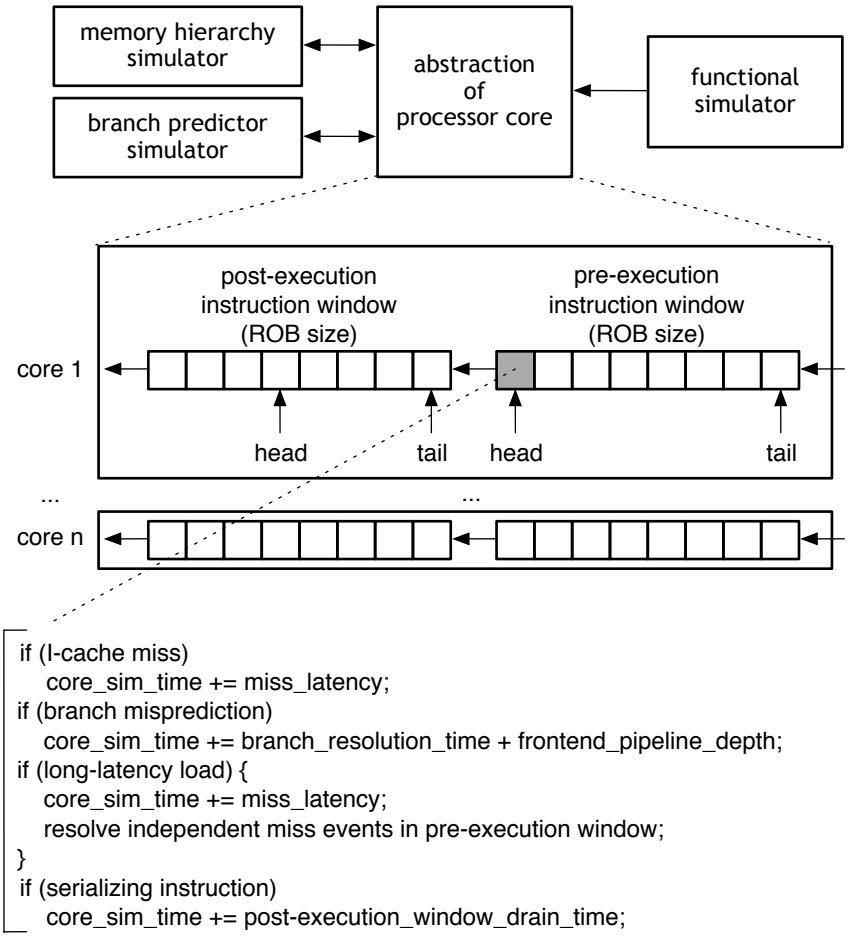

Figure 6.2: Schematic view of the interval simulation framework.

core simulated time by the branch resolution time and the front-end pipeline depth. In case of a long latency load (i.e., a last-level cache miss or cache coherence miss), we add the miss latency to the core simulated time, and we scan the pre-execution window for independent miss events (cache/TLB misses and branch mispredictions) that are overlapped by the long-latency load. For a serializing instruction, we add the window drain time to the simulated core time. If none of the above cases applies, we dispatch instructions at the effective dispatch rate.

Having determined the impact of the instruction at the pre-execution window head on the core's progress, we remove the instruction from the pre-execution window and feed it into the so called *post-execution window*. The post-execution window is used to derive the dependence chains of instructions and their impact on the branch resolution time and the effective dispatch rate in the absence of miss events, as we explain in detail in the following sections.

6.2.2 Interval simulation: detailed algorithm

We refer to the high-level pseudocode given in Figure 6.3 for a more detailed description of interval simulation. The interval simulator iterates across all cores in the multicore processor (line 2), and proceeds with the simulation as long as there are instructions to be simulated (line 3); if not, the simulator quits (line 69).

Multicore simulated time versus per-core simulated time

The interval simulator simulates cycle per cycle, and keeps track of the multicore simulated time as well as the per-core simulated time. The multicore simulated time is incremented every cycle (line 71). The per-core simulated time is adjusted depending on the progress of the individual core, e.g., in case of a miss event, the per-core simulated time is augmented by the appropriate penalty. Only in case the per-core simulated time equals the multicore simulated time, we need to simulate the cycle for the given core (line 6). In case the per-core simulated time is larger than the multicore simulated time (which can happen because of miss events as we will describe next), we do not need to simulate the cycle for the given core. This could be viewed as event-driven simulation at the core level.

```
while (1) {
|   for (i = 0; i < num_cores; i++) {
|   |   if (there are more insns to be simulated) {
|   |   |
|   |   |   insns_dispatched = 0;
|   |   |   while ( (core_sim_time [i] == multi_core_sim_time) &&
|   |   |   |       (insns_dispatched < eff_dispatch_rate(i) ) {
|   |   |   |
|   |   |   |   consider insn at pre-execution window head;
|   |   |   |
|   |   |   |   /* handle I-cache and I-TLB */
|   |   |   |   if (!I_overlapped) {
|   |   |   |   |   miss_latency = Icache_and_ITLB_access();
|   |   |   |   |   if (Icache_or_ITLB_miss) {
|   |   |   |   |       core_sim_time [i] += miss_latency;
|   |   |   |   |       empty_post-execution_window();
|   |   |   |   |   }
|   |   |   |   }
|   |   |   |
|   |   |   |   /* handle branch prediction */
|   |   |   |   if (branch && !br_overlapped) {
|   |   |   |   |   branch_predictor_access();
|   |   |   |   |   if (branch_misprediction) {
|   |   |   |   |       core_sim_time [i] += branch_resolution_time() +
|   |   |   |   |                            frontend_pipeline_depth;
|   |   |   |   |       empty_post-execution_window();
|   |   |   |   |   }
|   |   |   |   }
|   |   |   |
|   |   |   |   /* handle loads and stores */
|   |   |   |   if (store || (load && !D_overlapped)) {
|   |   |   |   |   miss_latency = Dcache_and_DTLB_access();
|   |   |   |   |   if (long_latency_load) {
|   |   |   |   |   |
|   |   |   |   |   |   for (all insns in pre-execution window from head to tail) {
|   |   |   |   |   |   |
|   |   |   |   |   |   |   I_overlapped = 1; I_cache_and_ITLB_access();
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   if (branch && independent of long-latency load) {
|   |   |   |   |   |   |   |       br_overlapped = 1; branch_predictor_access();
|   |   |   |   |   |   |   |       if (branch_misprediction) break;
|   |   |   |   |   |   |   }
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   if (load && independent of long-latency load) {
|   |   |   |   |   |   |   |        D_overlapped = 1; Dcache_and_DTLB_access();
|   |   |   |   |   |   |   }
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   if (serializing insn) break;
|   |   |   |   |   |   |   }
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   core_sim_time [i] += miss_latency;
|   |   |   |   |   |   |   empty_post-execution_window();
|   |   |   |   |   |   |   }
|   |   |   |   |   }
|   |   |   |   |   /* handle serializing insns */
|   |   |   |   |   if (serializing insn) {
|   |   |   |   |   |   core_sim_time [i] += empty_window_latency();
|   |   |   |   |   |   empty_post-execution_window();
|   |   |   |   |   }
|   |   |   |   |   /* dispatch insn */
|   |   |   |   |   insns_dispatched++;
|   |   |   |   |   insert_dispatched_insn_in_post-execution_window();
|   |   |   |   |   enter_new_insn_in_pre-execution_window();
|   |   |   |   }
|   |   |   |   if (core_sim_time [i] == multi_core_sim_time)
|   |   |   |       core_sim_time [i]++;
|   |   |   }
|   |   |   else
|   |   |       finish_simulation();
|   |   }
|   |   multi_core_sim_time++;
}
```

Figure 6.3: High-level pseudocode for multicore interval simulation.

Instruction dispatch

As long as no miss events have occurred in the given cycle (line 6) and the core has dispatched fewer instructions than the *effective dispatch rate* (line 7), we continue to simulate instructions. If we have exceeded the given effective dispatch bandwidth and no misses have occurred, we increment the per-core simulated time (lines 65–66). We will describe how we compute the effective dispatch rate further in this chapter.

The core-level simulation considers the instruction at the pre-execution window head (line 9) and determines its (potential) miss penalty (lines 11–59). Once we have computed the instruction's execution latency, we increment the number of dispatched instructions (line 61), and we move the instruction from the pre-execution window to the post-execution window (line 62). We subsequently insert a new instruction (supplied by the functional simulator) in the pre-execution window at the entry pointed to by the tail pointer (line 63).

Miss events

For each non-overlapped instruction we access the I-cache/TLB access (line 13). If this instruction is an I-cache miss or an I-TLB miss, we add the miss latency to the per-core simulated time (line 15). We will explain the purpose of lines 12 and 16 further in this chapter.

For non-overlapped branches we also access the branch predictor (line 22). The timing impact of a branch misprediction is fairly similar to an I-cache/TLB miss. If the branch is mispredicted (line 23), we add the branch penalty to the per-core simulated time. The branch penalty is computed as the sum of the branch resolution time and front-end pipeline depth (lines 24–25). We will explain how we estimate the branch resolution time later; the front-end pipeline depth is a known microarchitecture parameter.

For stores and non-overlapped loads (line 31), we access the memory hierarchy (i.e., caches including the cache coherence protocol, TLBs, interconnection network, and main memory) (line 32). In case of a long-latency load, we incur a miss penalty (i.e., the miss latency) which is added to the per-core simulated time (line 51).

Serializing instructions cause the core to drain the instruction window prior to their execution. Therefore, upon a serializing instruction, we increase the per-core simulated time with the latency for emptying

the post-execution instruction window (lines 56–59).

Overlapping miss events: second-order effects

A long-latency load, as mentioned in the previous section, may hide latencies by other subsequent independent miss events. Upon a long-latency load, we walk over the instructions in the pre-execution window from head to tail (line 35) and consider four cases (lines 35–49).

First, we access the I-cache and I-TLB (line 37). We mark the instruction as overlapped meaning that the I-cache/TLB access (a potential I-cache/TLB miss) is hidden by the long-latency load—this is done through the I_overlapped variable. This means that the I-cache/TLB access has occurred and should not incur any additional penalty when it appears at the pre-execution window head (line 12). In other words, the I-cache/TLB access/miss is hidden underneath the long-latency load—a second-order effect.

Second, for a branch that is independent of the long-latency load, we access the branch predictor and mark the branch as overlapped. When this branch appears at the pre-execution window head, we do not account for the potential misprediction penalty. Moreover, if this branch is mispredicted it will cause subsequent loads to serialize with the blocking miss at the pre-execution window head. We then break out of the loop and stop scanning the pre-execution window (lines 39–42).

Third, we access the memory hierarchy in case of an independent load, and mark the load as overlapped. We consider a load as independent if there are no data dependences between this load and the long-latency load at the pre-execution window head, and if there appears no memory barrier between the two loads in the dynamic instruction stream—a memory barrier between two loads will serialize their latencies. If the overlapped memory access results in a long-latency D-cache/TLB miss, we do not incur the miss penalty (lines 44–46). Moreover, this overlapped load miss is the head of a chain of dependent instructions, which will serialize with the load and thus serialize with the long-latency load at the pre-execution window head. Figure 6.4 gives a scenario where an independent load serializes with the long-latency load at the head of the ROB. The long-latency load (4) is independent of the long-latency load at the head of the ROB, but it serializes with the blocking miss because it depends on another long-latency load (2) that is overlapped by the load at the ROB head. Chen and Aamodt [13]

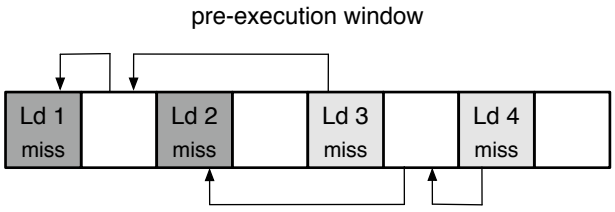

Figure 6.4: Independent long-latency loads that will (not) overlap with a blocking miss at the head of the ROB. All loads in this figure are long-latency loads. Load 2 is independent of the blocking load 1 at the head of the ROB, hence it is overlapped by load 1. Load 3 depends on load 1, thus it serializes with the blocking miss. Load 4 is independent on load 1, however, it depends on load 2, therefore, it serializes with the blocking miss.

describe a similar scenario where independent loads serialize with the blocking miss at the ROB head.

Fourth, in case we reach a serializing instruction, we break out of the loop and stop scanning the pre-execution window (line 48). The serializing instruction causes the window to be drained.

Branch resolution time, window drain time, and effective dispatch rate

An important component in interval simulation is to estimate the critical path length (including the latencies) in the post-execution window. The critical path length is used for computing (i) the branch resolution time (line 24), (ii) the window drain time upon a serializing instruction (line 57), and (iii) the effective dispatch rate (line 7). For computing the critical path length, we consider a very simple data flow model that computes the earliest possible *finish* time for each instruction in the post-execution window given its dependences and execution latency. The finish time is the cycle in which an instruction finishes its execution on a functional unit or the cycle when the result is retrieved from memory in case of a load; a dependent instruction can then execute in the next cycle.

Figure 6.5 shows a sequence of instructions with their dependences and execution latencies; the real critical path in the ROB is shown by bold arrows. We compute the critical path length as the post-execution window slides over the instruction sequence as follows.

For each instruction that is inserted at the post-execution window

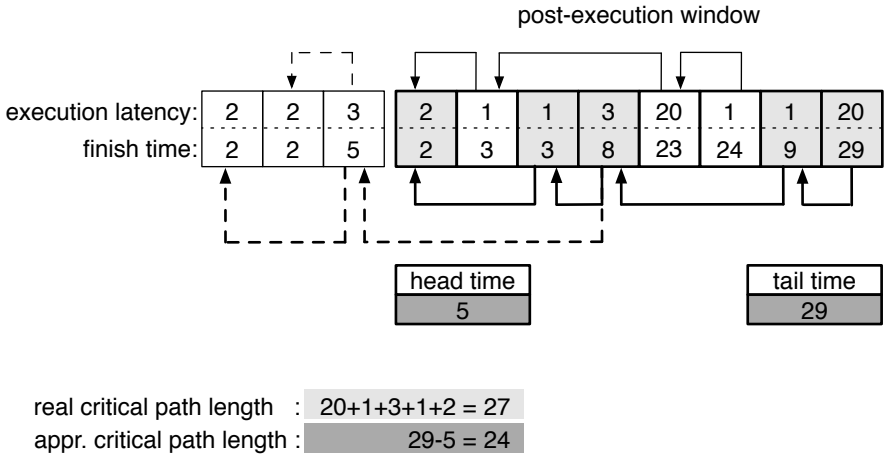

Figure 6.5: Illustrating critical path length computation during interval simulation.

tail, we compute its finish time as the maximum finish time of the instructions that it depends upon plus its execution latency. The execution latency equals the L2 cache hit latency in case of a short-latency load miss, and the functional unit latency for all other instruction types. For each instruction in the post-execution window we keep track of its input register dependences, and in case of a load we track its RAW memory dependence, i.e., the store instruction that accesses the same memory address. Furthermore, we also keep track of the cache line reuse dependences for memory references. This is to model the effect that delayed hits have on the critical path, i.e., delayed hits can not complete before their primary miss.

In addition, we keep track of the post-execution window's *head time* and *tail time*, see also Figure 6.5. The tail time is computed as the maximum of the tail time and the finish time of the newly inserted instruction at the tail of the post-execution window. Similarly, the head time is the maximum of the head time and the finish time of the removed instruction. We then *approximate* the length of the critical path in the post-execution window as the tail time minus the head time. Computing the real critical path in the ROB requires walking the post-execution window for every newly inserted instruction, which is too time-consuming. We found this approximation to be accurate enough for our purpose, as we will demonstrate in the evaluation, see Section 6.4 and more specifically Figure 6.6(a).

Our method measures the critical path over the full program trace, instead of measuring the critical path in the post-execution window. This can be seen in Figure 6.5; the critical path over the full trace is shown by the bold dashed arrows and the remaining critical path—in bold arrows—starting from the fourth instruction in the post-execution window—instruction with finish time = 8. The real critical path in the post-execution window may be (slightly) larger than the critical path observed through our method—the real critical path is 27 whereas our approximation is 24. The difference comes from an additional short dependency chain—the first and the third instruction in the post-execution window. Both instructions are part of the real critical path in the post-execution window, however they are not part of the critical path over the full trace. In other words, the instructions on the additional data dependency path are producers of an instruction on the critical path over the full trace—fourth instruction in the post-execution window. Furthermore, these instructions are not consumers of an instruction on the critical path over the full trace—already removed from the post-execution window. Larger data dependence distances on the critical path over the full trace will increase this effect. This error is more pronounced if the real critical path in the post-execution window is short. However, this is not a problem because in this case the effective dispatch rate is (most likely) larger than the dispatch width of the pipeline.

Note that the earliest possible finish time of a newly inserted instruction is not only determined by the maximum finish time of its producers, but also by the post-execution window's head time. The inserted instruction's finish time then equals its execution latency plus the maximum of the head time and the slowest producer's finish time.

Once we have computed the critical path length, we can compute the maximum possible execution rate through the post-execution window. Using Little's Law, we compute the execution rate as the window size divided by the critical path length. This reflects the fact that the out-of-order processor cannot process instructions faster than dictated by the critical path length. The effective dispatch rate then equals the minimum of this execution rate and the designed dispatch width. Similarly, the window drain time is computed as the maximum of (i) the number of instructions in the post-execution window divided by the processor's dispatch width, and (ii) the length of the critical execution path in the post-execution window.

The branch resolution time is computed as the longest chain of dependent instructions (including their execution latencies) leading to the mispredicted branch, starting from the head pointer in the post-execution window.

Interval length effect

Interval length (the number of instructions between two subsequent miss events) has a significant impact on overall performance. In particular for a mispredicted branch, a short interval implies a short dependence path to the branch (i.e., a short branch resolution time). A long interval on the other hand, implies a longer branch resolution time. A similar effect occurs for serializing instructions. A serializing instruction causes the instruction window to be drained. Window drain time is correlated with the interval length prior to the serializing instruction. In order to model the dependence of interval length on branch resolution time and window drain time, we empty the post-execution window upon a miss event (lines 16, 26, 52, and 58).

6.2.3 Limitations

Our current implementation of interval simulation employs a functional-first simulation approach. This means that the functional simulator generates a dynamic instruction stream, including user-level and system-level code, that is subsequently fed into the timing simulator. This implies that interval simulation does not simulate instructions along mispredicted paths, hence, it does not capture the positive or negative interference that wrong-path instructions may have on performance, e.g., through cache prefetching effects. Moreover, functional-first interval simulation may lead to different thread interleavings than what may happen in real systems.

A more accurate approach is to build a timing-directed simulator in which the timing simulator directs the functional simulator along mispredicted paths and determines thread interleavings. Unfortunately, timing-directed simulators are more difficult to develop. In our current implementation we opted for functional-first simulation because of its ease of development—this is a trade-off in development time, evaluation time and accuracy—and our evaluation shows good accuracy against the cycle-accurate simulator. Implementing timing-directed interval simulation may be a course of future work.

benchmark	input
blackscholes	<cores> 4096
bodytrack	sequenceB_1 4 1 1000 5 0 <cores>
canneal	<cores> 10000 2000 100000.nets
dedup	-c -p -f -t <cores> -i medias.dat -o output.dat.ddp
fluidanimate	<cores> 5 in_35K.fluid out.fluid
streamcluster	10 20 32 4096 4096 1000 none output.txt <cores>
swaptions	-ns 16 -sm 5000 -nt <cores>
vips	im_benchmark pomegranate_1600x1200.v output.v
x264	–quiet –qp 20 –partitions b8x8,i4x4 –bframes 3 –ref 5 –direct auto –b-pyramid –b-rdo –weightb –bime –mixed-refs –no-fast-pskip –me umh –subme 7 –analyse b8x8,i4x4 –threads <cores> -o eledream.264 eledream_640x360_8.y4m

Table 6.1: The multithreaded PARSEC benchmarks and their reference inputs used in the full-system simulation experiments.

An obvious limitation of interval simulation is that it does not model core-level performance in a detailed cycle-accurate way. Again, this choice is motivated by our goal to improve simulator development time and evaluation time.

Another limitation is that the timing of miss events may be different compared to real systems. In particular, interval simulation walks the pre-execution instruction window upon a long-latency load and simulates all miss events that are independent of the long-latency load in the same cycle, which may be different from what may happen in a real system. This is a reasonable assumption, because the miss events that are independent of the long-latency load are hidden anyway.

6.3 Experimental setup

We use two benchmark suites, namely SPEC CPU2000 and PARSEC [5]. We use all of the SPEC CPU2000 benchmarks with the reference inputs in our experimental setup, see Table 4.1. The binaries of the CPU2000 benchmarks were taken from the SimpleScalar website; these binaries were compiled for Alpha using aggressive compiler optimizations. We considered one hundred million simulation points as determined by SimPoint [67] in all of our experiments in order to limit overall cycle-accurate simulation time—this is exactly the problem tackled by inter-

Processor core	
ROB	256 entries
issue queue	128 entries
load-store queue	128 entries
store buffer	64 entries
processor width	decode, dispatch and commit 4 wide issue 6 wide fetch 8 wide
functional units	4 integer, 4 load/store and 4 floating-point
functional unit latencies	load (2), mul (3), fp (4), div (20)
fetch queue	16 entries
front-end pipeline depth	7 stages
branch predictor	12 Kbit local predictor, 8-way set-assoc 2 K-entry BTB, 32-entry RAS
Memory subsystem	
L1 I-cache	32 KB 4-way set-assoc 64 B line size
L1 D-cache	32 KB 4-way set-assoc 64 B line size
L2 cache	unified, 4 MB 8-way set-assoc 64 B line size, 12 cycles access latency
coherence protocol	MOESI
main memory	150 cycle access time
memory bandwidth	10.6 GB/s peak bandwidth

Table 6.2: Baseline processor core model assumed in our experimental setup; simulated CMP architectures share the L2 cache.

val simulation. When simulating multiprogram workloads, we stop simulation as soon as one of the co-executing programs has executed one hundred million instructions.

In addition to the single-threaded user-level SPEC CPU benchmarks, we also use the multithreaded PARSEC benchmarks which spend a substantial fraction of their execution time in system code. We use 9 of the 13 PARSEC benchmarks that run on our simulator with the small input set, see Table 6.1, and run each benchmark to completion; the number of dynamically executed instructions per benchmark varies between five hundred million to thirteen billion instructions. The PARSEC benchmarks were compiled using the GNU C compiler for Alpha; we use aggressive optimization, including `-O3`, loop unrolling and software prefetching.

We use the M5 simulator [6] in all of our experiments. The SPEC CPU benchmarks are run in user-level simulation mode, and the PAR-

SEC benchmarks are run in full-system simulation mode.

Our baseline core microarchitecture is a 4-wide superscalar out-of-order core, see Table 6.2. When simulating a CMP, we assume that all cores share the L2 cache as well as the off-chip bandwidth for accessing main memory, and we assume a MOESI cache coherence protocol. For the multiprogram workloads, we were able to simulate up to 8 cores; physical memory constraints limited us from running larger multicore processor configurations. For the multithreaded workloads, we also run up to 8 cores; the benchmarks run on top of the 2.6.8.1 Linux kernel.

6.4 Evaluation

We now evaluate interval simulation in terms of development cost, accuracy and simulation speed. Accuracy is evaluated through a number of experiments, and we consider single-threaded workloads, homogeneous and heterogeneous multiprogram workloads, multithreaded workloads, and a performance trend case study.

6.4.1 Code size of core-level model

As mentioned earlier, an important goal for interval simulation is to reduce the complexity of architectural simulators by raising the level of abstraction. The detailed cycle-level out-of-order processor model used in our experiments is about twenty eight thousand lines of code, whereas the core-level mechanistic analytical model is less than one thousand lines of code. Thus, interval simulation is easy to implement and reduces the development cost drastically.

6.4.2 Single-threaded

We first consider single-threaded workloads running on a single-core processor, and evaluate interval simulation in a step-by-step manner in order to understand where the error sources are. For doing so, we consider the following experiments; each experiment evaluates a particular aspect of interval simulation:

- *Effective dispatch rate*: We consider the branch predictor to be perfect (i.e., all branch predictions are correct), as well as the I-cache/TLB and L2 cache (i.e., all cache accesses are hits). The

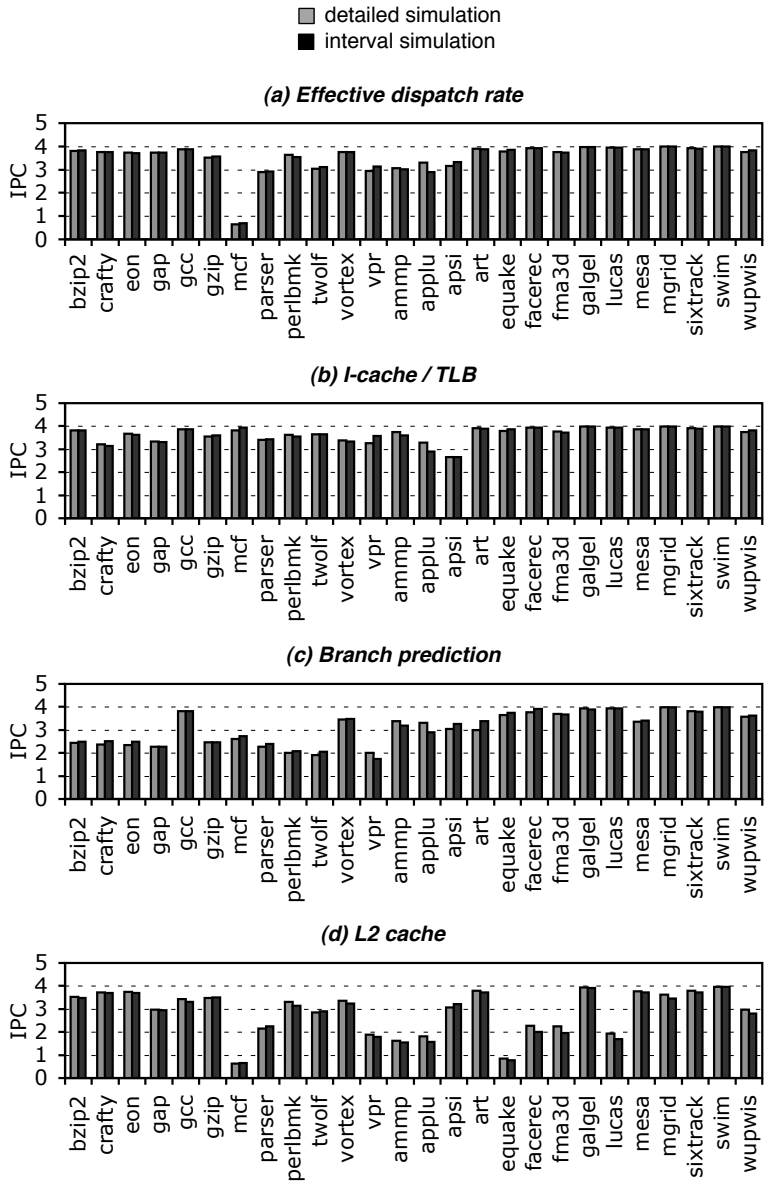

Figure 6.6: Evaluating interval simulation in a step-by-step manner: evaluating the modeling accuracy of the (a) effective dispatch rate, (b) I-cache/TLB, (c) branch prediction and (d) L2 cache.

L1 D-cache is non-perfect. This setup aims at evaluating the accuracy of the modeling of the effective dispatch rate.

- *I-cache/TLB*: The branch predictor is perfect as well as L1 and L2 D-cache and D-TLB. The I-cache and I-TLB are non-perfect.
- *Branch prediction*: All caches are assumed to be perfect. The only non-perfect structure is the branch predictor.
- *L2 cache*: The L1 I-cache is assumed to be perfect as well as the branch predictor. The L1 D-cache and L2 cache are non-perfect.

Figure 6.6 compares the IPC measured through detailed simulation versus the IPC estimated through interval simulation for each of the above four experiments. Figure 6.6 (a) and (b) shows that the effective dispatch rate and I-cache/TLB behavior is modeled accurately: the average error for both experiments is 1.8%. We observe slightly higher errors for the branch prediction and L2 cache modeling with average errors of 3.8% and 4.6%, respectively, see Figure 6.6(c) and (d). The difficulty in predicting the impact of branch mispredictions on performance is due to estimating the branch resolution time. The branch resolution time is the number of cycles between the mispredicted branch being dispatched in the instruction window and the branch being resolved, or in other words, the critical path leading to the mispredicted branch (due to the instructions that are yet to be executed in the instruction window). Interval simulation however approximates the branch resolution time by the critical path leading to the mispredicted branch in the post-execution window. This is an overestimation of the penalty if the critical path is partially executed by the time the mispredicted branch enters the instruction window, or is an underestimation if critical path execution gets slowed down because of resource contention. With respect to estimating the performance impact of L2 cache misses, interval simulation tends to (slightly) overestimate the penalty due to L2 misses. Interval simulation basically assumes there are no instructions dispatched underneath the L2 miss, however, the processor may be dispatching instructions while the L2 miss is being resolved.

6.4.3 Homogeneous workloads

Putting everything together, the average error for the single-threaded benchmarks equals 5.9%, see the top graph in Figure 6.7. The other

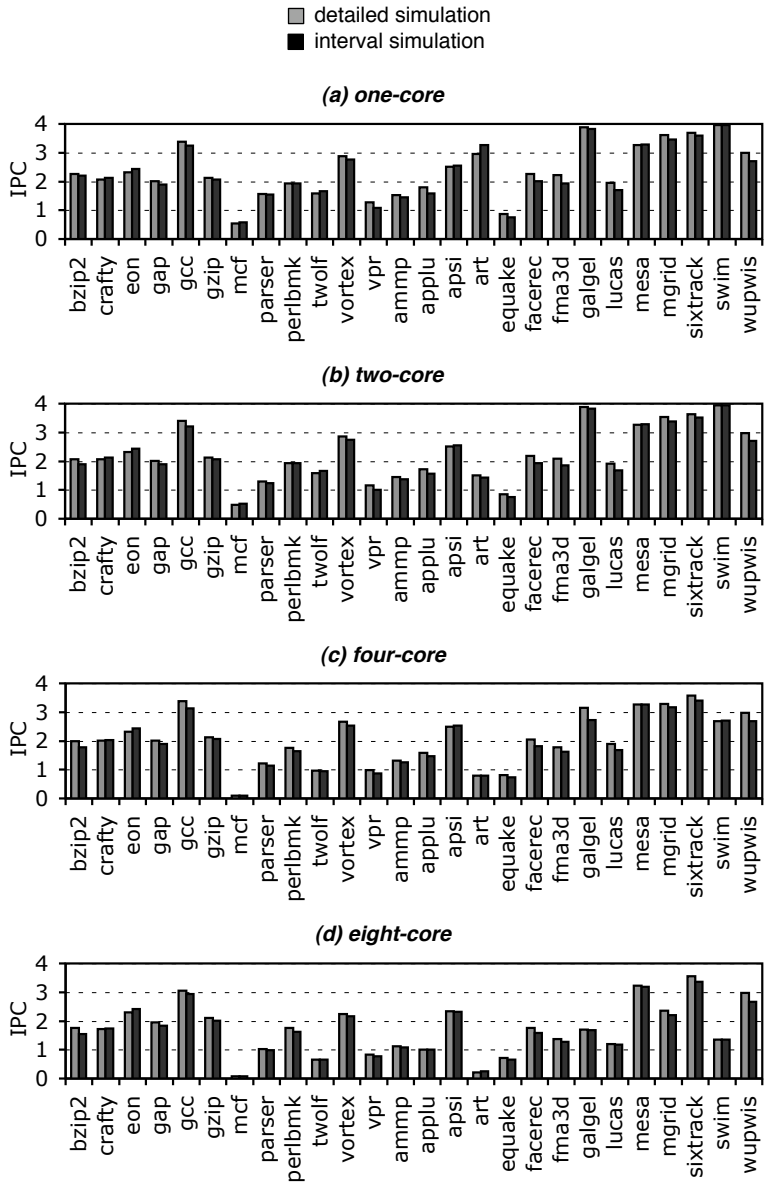

Figure 6.7: Evaluating the accuracy of interval simulation for the single-threaded SPEC CPU benchmarks. The two-core, four-core and eight-core experiments assume two, four and eight copies of the same benchmark, respectively.

three graphs in Figure 6.7 show that interval simulation is capable of tracking the conflict behavior in the shared L2 cache when running homogeneous multiprogram workloads on a multicore processor (i.e., multiple copies of the same program run concurrently). The average IPC prediction error for the two-core, four-core and eight-core machines are 5.9%, 6.0% and 4.8%, respectively. The largest errors are due to estimating the branch prediction penalty (see vpr, applu, art), and the L2 cache/TLB miss penalty (see equake, facerec, fma3d and lucas). The maximum error observed in the design space is limited to 15.5% (for vpr on a single-core processor). Interval simulation clearly identifies which benchmarks are susceptible to L2 cache sharing. For example, the average IPC for art gradually drops as it is co-executed with a copy of itself on each core of a two-core, four-core and eight-core CMP. The average IPC for swim, and less pronounced for mgrid, drops with a four-core and eight-core CMP. Some benchmarks are not affected by L2 cache sharing as shown by both detailed and interval simulation, e.g., mesa, sixtrack and wupwise.

6.4.4 Heterogeneous workloads

The next step in our evaluation considers heterogeneous multiprogram workloads, i.e., multiple single-threaded workloads co-executing on a multicore processor in which each core executes one single-threaded workload. Figure 6.8 evaluates the accuracy for four heterogeneous multiprogram workloads in terms of STP and ANTT. In each graph there are three bars for each benchmark, corresponding with a two-core, four-core and eight-core CMP. For example, see for the leftmost workload mix in Figure 6.8, we co-run art with applu, mcf with lucas, equake with wupwise, and swim with facerec on a two-core configuration; for the four-core configuration, we co-run art, applu, mcf and lucas, and we co-run equake, wupwise, swim and facerec; for the eight-core configuration we co-run all benchmarks in the workload.

The average error observed across all heterogeneous workloads equals 2.0% and 2.9% for STP and ANTT, respectively; the maximum error is 5.7% and 8.6% (STP and ANTT for mix 4 on 8 cores). The important observation from Figure 6.8 is that interval simulation tracks performance trends very accurately, and is capable of modeling conflict behavior in shared last-level caches. For example, we observe that STP improves less for memory-intensive workloads (e.g., mix 1) compared to computation-intensive workloads (e.g., mix 3), which suffer less

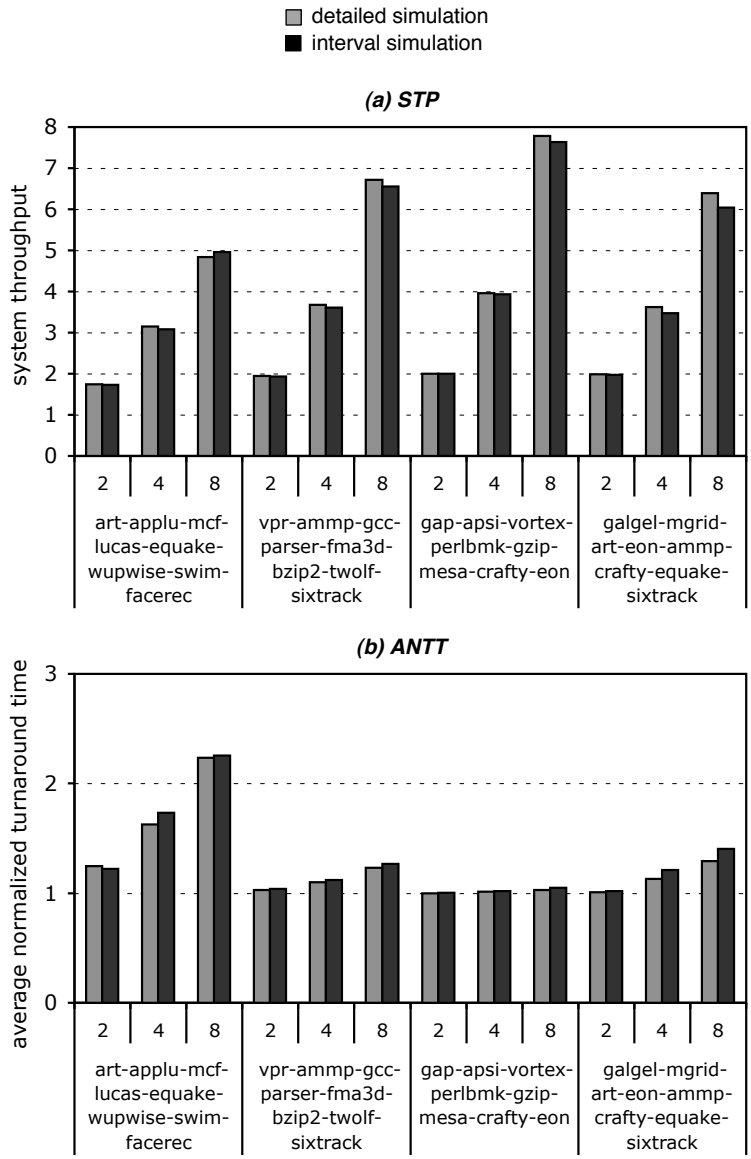

Figure 6.8: Evaluating the accuracy of interval simulation for four heterogeneous multiprogram workloads in terms of (a) STP and (b) ANTT as a function of the number of cores. The two-core configuration results show STP/ANTT for workload mixes of co-executing benchmarks; e.g., in the leftmost workload mix, we co-run art with applu, mcf with lucas, etc. For the four-core configuration, we co-run art, applu, mcf and lucas, and we co-run equake, wupwise, swim and facerec; for the eight-core configuration we co-run all benchmarks in the workload mix.

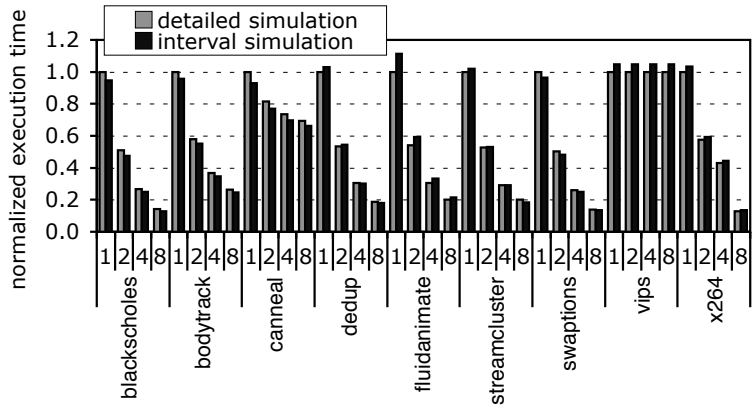

Figure 6.9: Evaluating the accuracy of interval simulation for the multithreaded full-system PARSEC workloads as a function of the number of cores. Performance numbers are normalized to detailed cycle-accurate single-core simulation.

from conflict misses due to cache sharing in a CMP. The same observation can be made for ANTT; ANTT is not affected very much for mix 2, 3 and 4, however, it drastically increases for mix 1.

6.4.5 Multithreaded workloads

We now consider the multithreaded PARSEC benchmarks. Figure 6.9 shows normalized execution time as a function of the number of cores that the multithreaded workload runs on. The average error when comparing the estimated execution time obtained through interval simulation versus cycle-accurate simulation is 4.6%: the error is below 6% for most benchmarks, except for fluidanimate (11.4%). The important observation is that interval analysis estimates the performance trend with the number of cores accurately. For example for vips, interval simulation accurately tracks that performance does not improve with an increasing number of cores. The fact that performance does not scale with the number of cores is due to load imbalance and poor synchronization behavior. For the other benchmarks, performance improves with an increasing number of cores. Interval simulation tracks this trend accurately, in spite of the absolute error, even for fluidanimate.

6.4.6 Cache design space exploration

Figure 6.10 illustrates the accuracy of interval simulation for exploring the shared L2 cache design space. We vary the size of the shared L2 cache (1 MB, 2 MB, 4 MB and 8 MB) and its set-associativity (4, 8 and 16). Each graph shows the normalized execution time; performance numbers are normalized to detailed cycle-accurate single-core simulation with a 1 MB 4-way set associative L2-cache. We show the average normalized execution time over all PARSEC benchmarks as well as the normalized execution time for canneal, vips and streamcluster running on a two-core and four-core CMP. We obtained similar results for other benchmarks. The overall conclusion is that despite the absolute error, interval simulation tracks performance differences across multiple last-level cache configurations and across a different number of cores.

6.4.7 Performance trend case study

Our case study considers a performance trade-off as a result of 3D stacking [48], and compares two processor architectures. Our first processor architecture is a dual-core processor with a 4 MB L2 cache which is connected to external DRAM through a 16-byte wide memory bus; our second processor architecture is a quad-core processor which is connected to integrated DRAM (through 3D stacking) through a 128-byte memory bus and which does not have an L2 cache. External DRAM is assumed to have a 150-cycle access latency; 3D-stacked DRAM is assumed to have a 125-cycle access latency.

The important observation from Figure 6.11 is that interval simulation leads to the same conclusions as detailed cycle-accurate simulation. The quad-core processor leads to better performance for a number of benchmarks, such as bodytrack, fluidanimate and swaptions; for the other benchmarks on the other hand, cache space is more important than processing power, and hence, the dual-core processor outperforms the quad-core processor, see canneal, vips and x264. This case study illustrates that interval simulation leads to the same conclusions in practical design trade-offs.

6.4.8 Simulation speed

Interval simulation is substantially faster than detailed cycle-level simulation, see Figure 6.12, which shows the simulation speedup through

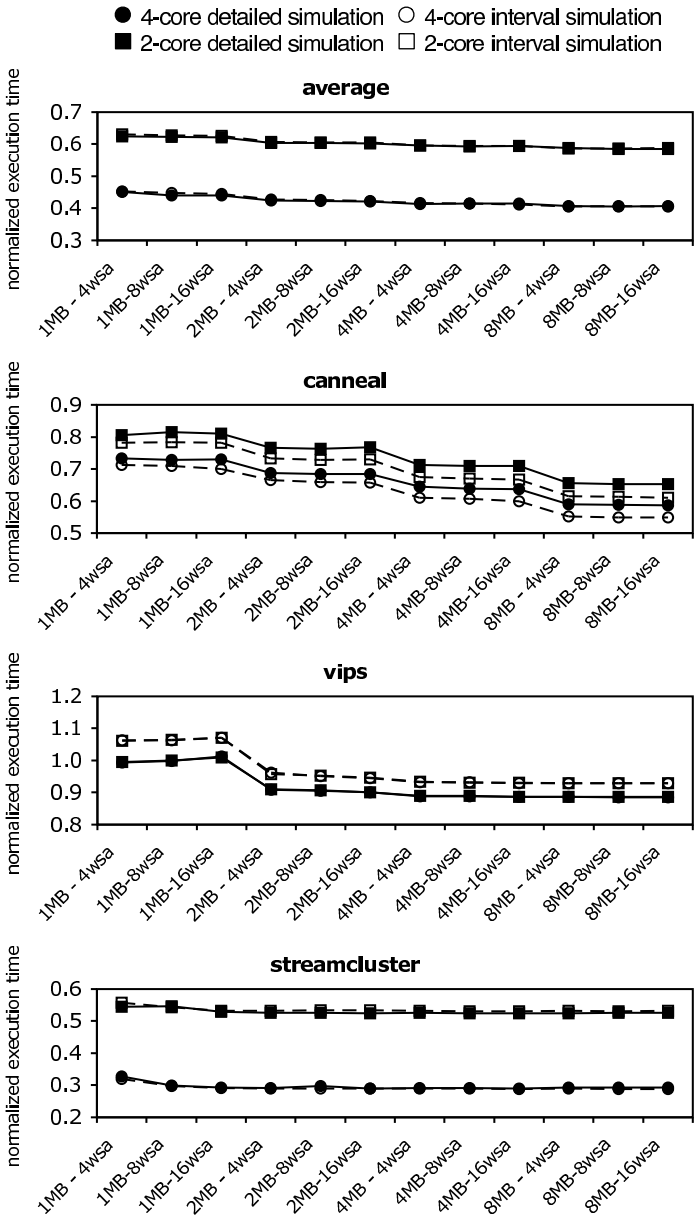

Figure 6.10: Evaluating the accuracy of interval simulation for tracking shared cache performance as a function of the cache configuration and the number of cores. Performance numbers are normalized to detailed cycle-accurate single-core simulation with a 1 MB 4-way set associative L2-cache.

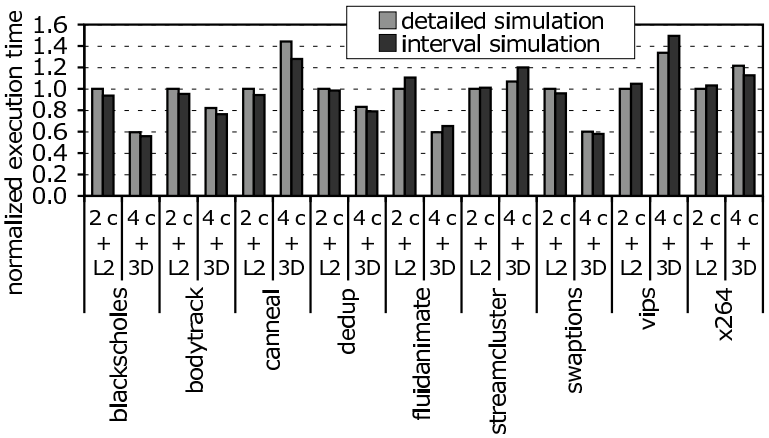

Figure 6.11: Evaluating interval simulation in a practical design trade-off: a dual-core processor with 4 MB L2 and external DRAM versus a quad-core processor with 3D-stacked DRAM and no L2 cache. Performance numbers are normalized to detailed simulation of the dual-core processor configuration.

interval simulation compared to detailed simulation for the multiprogram workloads and multithreaded workloads. The average simulation speedup is a factor 10× for the multithreaded workloads and 4 cores, and a factor 14× for the multiprogram workloads and 8 cores. We observe that higher simulation speedups are achieved for memory-intensive benchmarks, see for example mcf and art, because of the *functional* implementation of the memory subsystem during interval simulation in contrast to the event-based simulation under detailed simulation.

6.5 Related Work

As mentioned before, computer architects rely heavily on cycle-level simulators. Developing cycle-accurate simulators is very time-consuming and in addition, they have relatively poor performance in terms of simulation speed. Development effort and evaluation time are becoming a major concern as the number of cores on a chip-multiprocessor increases. For this reason, it is not uncommon that architects make simplifying assumptions when simulating large multicore and multiprocessor systems.

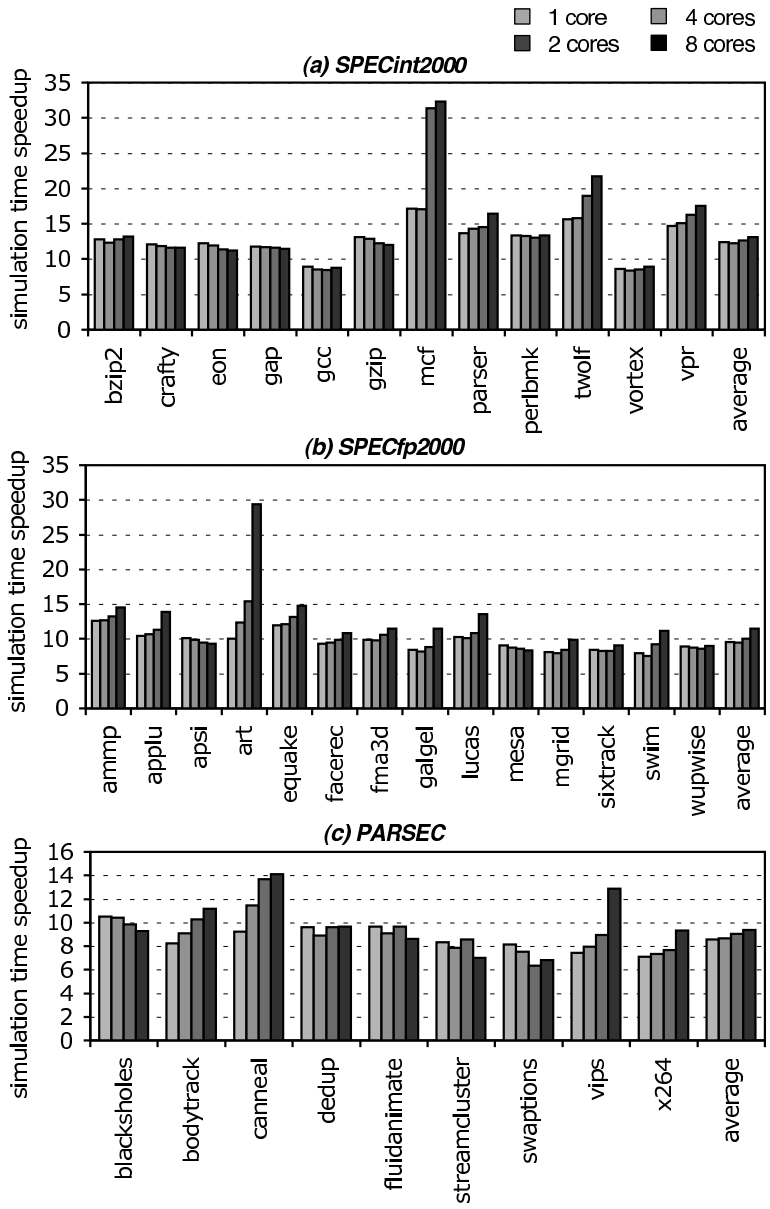

Figure 6.12: Simulation speedup compared to detailed cycle-accurate simulation for (a and b) the multiprogram SPEC CPU2000 workloads and (c) the multithreaded PARSEC benchmarks.

A common assumption is to assume that all cores execute—in program order—one instruction per cycle as long as no cache miss events occur, see for example [33, 42, 55]. Upon a miss event an additional penalty is added to the simulated clock—this could be called 'stall event' simulation. This approach does not model MLP—cache misses do not overlap. Neither does it accurately capture the relative progress (and the conflict behavior) between co-executing threads.

Interval simulation can be viewed along this line of 'stall event' simulation approaches, however, it captures the relative progress among threads more accurately. In the absence of miss events, interval simulation models the effective dispatch rate through the computation of the critical path in the ROB. When a long latency load miss event occurs, interval simulation models the MLP by walking over the pre-execution window and marking overlapped independent loads. In addition, interval simulation models branch prediction, which to our understanding is not modeled in the simulation methodologies used in [33, 42, 55]. Therefore, interval simulation is a fast and more accurate alternative for the one-IPC performance model.

6.6 Summary

In this chapter we have proposed interval simulation which raises the level of abstraction in architectural simulation. Interval simulation replaces the core-level cycle-accurate simulation model by a mechanistic analytical model. The analytical model estimates core-level performance by dividing the execution in so called intervals. The intervals are separated by miss events, i.e., branch mispredictions, TLB misses and cache misses (e.g., conflict misses, coherence misses, etc.); the miss events and their latencies are derived through simulation. By analyzing the types of miss events and their latencies, interval simulation can estimate core-level performance. The core-level analytical performance models drive the timing of the miss events, which in their turn determine core-level performance.

We evaluated the accuracy of interval simulation for both multiprogram SPEC CPU2000 workloads and multithreaded PARSEC benchmarks, running in user-level simulation mode and in full-system simulation mode, respectively. We report an average error of 5.9% for the single-threaded SPEC CPU2000 benchmarks compared to cycle-accurate simulation in M5, and an average error of 4.6% for the mul-

tithreaded PARSEC benchmarks. Interval simulation achieves a simulation speedup of one order of magnitude compared to cycle-accurate simulation. Moreover, interval simulation is easy to implement: our implementation of the core-level mechanistic analytical model is about one thousand lines of code, which is a dramatic reduction compared to a detailed cycle-level out-of-order processor simulation model (28 K lines of code for the out-of-order core model in M5). Thus, interval simulation tackles the simulator design trade-offs on two fronts: it reduces development and evaluation time. We believe that interval simulation is widely applicable for design studies that do not need cycle-accurate timing at the core level, e.g., when making design decisions in early stages of the design or when making system-level design trade-offs.

Chapter 7

Conclusion

Truth only reveals itself when one gives up all preconceived ideas.
Shoseki
The future influences the present just as much as the past.
Friedrich Nietzsche

In this dissertation, we have studied two simulation techniques, namely statistical simulation and interval simulation. Statistical simulation as presented in Chapter 3 forms the basis for the work discussed in Chapters 4 and 5. In Chapter 6 we have elaborated on interval simulation. In this chapter, we will summarize and compare both techniques. We will end with a discussion on future work.

7.1 Summary

Designing a new microprocessor is extremely time-consuming because of the large number of simulations that need to be run during design space exploration. On top of that, every single simulation run takes days or even weeks especially if complete benchmarks need to be simulated, i.e., architectural simulation is several orders of magnitude slower than real hardware execution. Recent technology trends, which put more and more cores on a single chip, exacerbates the simulation problem. Thus, the need arises for faster simulation techniques.

7.1.1 Statistical simulation

Statistical simulation is a fairly recently introduced simulation paradigm that reduces the simulation time, and thus can help to reduce the design cost of new microprocessors. It is a very fast simulation technique that only requires on the order of a million instructions per benchmark to make an accurate performance estimate. Therefore, it is a useful tool to cull a huge design space in limited time; a small region of interest identified through statistical simulation can then be further analyzed through more detailed and slower simulation runs.

The state-of-the-art in statistical simulation considered simple memory data flow models. We have proposed a more detailed and accurate modeling of the memory data flow in order to improve the accuracy of statistical simulation; we have modeled delayed hits, RAW memory dependences and cache miss correlation. Our experimental results have shown that accurately modeling memory data flow characteristics significantly reduces performance prediction errors, e.g, the average IPC prediction error has dropped from 10.9% down to 2.1%. Furthermore, statistical simulation accurately predicts performance trends, which makes it a useful tool for design space explorations.

The shift towards CMPs requires adjustments to statistical simulation in order to capture the conflict behavior in shared resources, such as last-level caches and main memory. We have shown that enhanced statistical simulation can predict this conflict behavior when running multiprogram workloads on a CMP. In order to do so, we have extended the statistical simulation paradigm in two ways. First, we have modeled time-varying execution behavior in the synthetic traces. Second, we have modeled cache behavior in a microarchitecture-independent way through memory reuse distance and stack depth distributions. Our experimental results have shown that statistical simulation is accurate with average IPC prediction errors of less than 7.3% over a broad range of CMP design points, while being one order of magnitude faster than detailed simulation. This makes statistical simulation a viable fast simulation approach to CMP design space exploration.

7.1.2 Interval simulation

We have introduced yet another novel simulation paradigm, namely interval simulation, which raises the level of abstraction in architectural

simulation. Interval simulation replaces the core-level cycle-accurate simulation model by a mechanistic analytical model. Instead of tracking the individual instructions as they propagate through the pipeline, it considers the streaming of instructions through the dispatch stage, in order to drive the timing simulation.

The streaming of instructions can be divided in so called intervals; each interval is marked by a miss event leading to the discontinuation of dispatching of instructions. The front-end will stop delivering useful instructions to the back-end in case of an I-cache/TLB miss, a branch misprediction or a serializing instruction. A long-latency load will cause dispatch to stall, i.e., the load will block commit once it is at the head of the ROB, the ROB fills up and eventually dispatch stops. Moreover, the long-latency load will overlap independent future miss events if both instructions causing the miss events reside in the ROB at the same time. Interval simulation can estimate core-level performance by analyzing the type of the miss events and their latencies. In turn, the estimated core-level performance drives the timing of (future) miss events.

Our experimental results have shown that interval simulation is fairly accurate; we have reported an average error of 4.6% for the multithreaded PARSEC benchmarks running in full-system simulation mode. Moreover, our results have demonstrated that interval simulation leads to correct design decisions in practical design studies. We have achieved a simulation speedup of one order of magnitude compared to cycle-accurate simulation. And finally, interval simulation is easy to implement; our model has required no more than one thousand lines of code, whereas a fully detailed simulator can easily consist of tens of thousand lines of code. We believe that interval simulation makes a good balance between development time, simulation speed and accuracy, especially when we do not need cycle-accurate timing at the core level.

7.2 Discussion

Computer architects heavily rely on cycle-level simulators, used at various stages of the design of a new processor. Past and current trends in processor technology (e.g., superscalar out-of-order execution and CMPs) increased the complexity of the microarchitecture, and thus the complexity of the simulators. This phenomenon makes truly cycle-

accurate simulation impractical for culling large design spaces. Therefore, researchers came up with various techniques to cope with the simulation problem.

Statistical simulation and interval simulation are two techniques that tackle this problem. Statistical simulation reduces the simulation time by reducing the number of instructions that need to be simulated, whereas interval simulation reduces the simulation time by raising the level of abstraction, in other words, reducing the number of instructions that must be executed per simulated instruction. Therefore, both techniques have their place in the computer architect's toolbox. As mentioned before these two techniques are orthogonal to and can be used in conjunction with each other as well as with other simulation speedup approaches, such as, sampled simulation, FPGA-accelerated simulation, etc.

We like to make the statement that we do not view these techniques as a replacement for cycle-accurate simulation. Instead, we view them as useful complements to cycle-accurate simulation and other tools that a computer designer has at his/her disposal for design studies where the simulation speed is a bigger issue than accuracy.

We will now compare both techniques in terms of accuracy, simulation speed and development effort. Furthermore, we will elaborate on the size of the statistical profiles in statistical simulation.

7.2.1 Accuracy

Both techniques are highly accurate; the average IPC prediction error for a single-core processor configuration is 2.4% for statistical simulation and 5.9% for interval simulation. Moreover, both techniques can capture the conflict behavior in shared caches when considering multiprogram workloads running on a CMP. For example, we achieve an average IPC prediction error of 7.3% for statistical simulation and 4.8% for interval simulation, when running homogeneous workloads on an eight-core CMP. More important when culling large design spaces is the relative error, which expresses the prediction error of one processor configuration relative to another configuration. Both techniques accurately track the performance trends over the various processor configurations, i.e., we achieve small relative errors. For example, our two techniques are able to accurately track performance differences between the design alternatives in a practical design study (3D stacking

case study).

7.2.2 Simulation speed

In our evaluation we have reported a simulation speedup of one order of magnitude for both simulation paradigms. However, some remarks are in place.

First, in Chapter 4 we have shown that statistical simulation for single-core processors can achieve a speedup of four orders of magnitude—we considered ten billion instruction traces for which we generated synthetic traces of one million instructions. On the other hand, when considering multicore processors, we need longer synthetic traces in order to obtain converged performance estimates, because the memory behavior in multicore statistical simulation is modeled through virtual address distributions, stack depth distributions and reuse distance distributions, rather than through cache miss rates. In our experiments in Chapter 5 we generated synthetic traces of ten million instructions, which is a factor ten smaller than the original one hundred million instruction SimPoint traces. Due to the huge memory requirements when collecting the statistical profiles, we were unable to perform a similar experiment (with ten billion instruction traces) as was done in Chapter 4. As a result, we were unable to show whether statistical simulation for multicore processors can achieve higher speedup factors for longer (ten billion instruction) traces.

Second, statistical simulation requires that we first collect a statistical profile, which is then used to generate a synthetic trace. This profile partially depends on the microarchitectural configuration. Hence, we have to recompute the statistical profile when we study a design space that varies a parameter on which the profile depends. Interval simulation, on the other hand, does not require an offline (profiling) phase. When we compare the evaluation time of statistical simulation against interval simulation, we also must take into account how much time is needed to collect the profile and to generate a synthetic trace from it. However, we improved the applicability of statistical simulation in Chapter 5 by making the profile less dependent on the microarchitecture, i.e., in contrast to single-core statistical simulation we modeled the caches in a quasi microarchitecture-independent way.

7.2.3 Development cost

The development time of a simulator is another important issue, that we address in this dissertation. Interval simulation solves this issue by raising the level of abstraction at the core-level, i.e., the simulator does not implement the core's pipeline in detail. Instead, it uses an analytical model to determine the rate at which instructions stream through the pipeline. The code size of the analytical model is much smaller compared to a typical detailed implementation of the pipeline. Thus, it is easier to implement and less error prone.

Statistical simulation, on the other hand, models the microarchitecture in a cycle-accurate way. Moreover, the synthetic trace simulator is a slightly modified version of a truly cycle-accurate simulator—the simulator is modified in order to deal with synthetic instruction traces rather than real instruction traces or executables. Furthermore, we also need to implement a profiling tool and a synthetic trace simulator. Therefore, the effort to develop a multicore processor statistical simulator is rather high: not as high as developing a truly cycle-accurate simulator, but still higher than interval simulation.

7.2.4 A note on the statistical profiles

As mentioned before, in Chapter 5 we have tried to compute the profiles of program traces of ten billion instructions. However, the amount of memory required during this profiling step was extremely high. As a consequence we were unable to collect the profiles on our 64-bit AMD machine with 8 GB of physical memory. The main reason why the memory usage during the profiling step is so high is that the profile contains virtual memory address distributions. These distributions are needed for the modeling of memory accesses during synthetic trace simulation, i.e., the address determines the accessed set in a cache. The physical memory requirement in order to be able to run the profiling tool on long instruction sequences is a limitation of CMP statistical simulation.

7.3 Future Work

In this section we will give some possible avenues along which to take future research.

7.3.1 Statistical simulation

Statistical simulation as described in Chapter 5 only considers multiprogram workload. Extending this technique to multithreaded workloads would improve the applicability for CMP statistical simulation. Therefore, we need to collect a distribution of address space identifiers in order to model sharing of memory addresses among co-executing threads—in the current implementation we use thread identifiers to distinguish between the memory spaces. In addition, combining it with the Nussbaum and Smith [60] and Hughes and Li [38] approaches will make statistical simulation viable for modeling multithreaded workloads running on CMPs with shared resources.

Second, this thesis also improves upon prior work by considering more realistic DRAM models. Statistical simulation could be further improved by considering more detailed cache models including hardware prefetching. Another interesting feature would be the modeling of caches that are optimized for miss patterns per static load or per group of loads. This may require a more accurate modeling of the local history of memory access patterns in addition to the global history described in Chapter 4.

Third, we have made a first but important step towards making the statistical profile microarchitecture-independent. The cache statistics are independent of the number of sets in the cache and the cache's associativity; they still depend on the cache line size though. The branch prediction statistics also depend on a particular branch predictor configuration. Making the statistical profile completely microarchitecture-independent would make the framework even more efficient and applicable. For example, statistically simulating SMT processors would then become plausible.

Fourth, we found the accuracy of the shared cache performance estimation through statistical simulation to be subject to warm-up in the shared cache. For now, we assume a hit-on-cold strategy. Considering potentially more accurate and efficient cache warm-up strategies may improve the overall prediction accuracy and lead to shorter synthetic traces.

7.3.2 Interval simulation

Our current implementation of an interval simulator employs a functional-first simulation approach, which may lead to different lock contention and different thread interleavings than what may happen in real systems. Implementing a timing-directed approach will capture lock contention and thread interleaving more accurately. Furthermore, simulating instructions along mispredicted paths would allow us to capture the interference that wrong-path instructions may have on performance. However, this will make interval simulation harder to develop—one must make a trade-off between accuracy and development effort.

Second, in this dissertation we have shown that interval simulation can accurately predict the performance of CMPs with up to eight cores. The next step for interval simulation is to consider many-core processors with multiple tens of cores. Hereto, we need to parallelize the interval simulator in order to simulate all cores concurrently. The simulation time can also be reduced through FPGA acceleration. The code size of interval simulation is much smaller compared to cycle-accurate simulation, i.e., 1 K vs. 28 K lines of code. Thus, interval simulation would allow us to place more cores on a single FPGA, which reduces the possible communication overhead between the FPGAs when simulating very large scale multicore systems.

Third, a possible application of interval simulation is to combine it with sampled simulation for CMPs. Single-core sampled simulation uses functional simulation as fast-forwarding strategy between sampling units. However, this would introduce inaccuracies in case of multicore processors, i.e., during fast-forwarding the relative progress of the threads would be 1 on 1, which is not the case in real systems. As illustrated in this dissertation, the relative progress between co-executing threads affects the conflict behavior in shared resources. Interval simulation is a potentially better solution as a fast-forwarding strategy between sampling units, because it captures the relative progress of the threads more accurately.

Bibliography

[1] Standard Performance Evaluation Corporation (SPEC). website. http://www.spec.org/.

[2] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani. UNISIM: An open simulation environment and library for complex architecture design and collaborative development. *IEEE Computer Architecture Letters*, 6(2):45–48, 2007.

[3] R. H. Bell, Jr. and L. K. John. Improved automatic testcase synthesis for performance model validation. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 111–120, 2005.

[4] E. Berg and E. Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *ISPASS '04: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 20–27, 2004.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[6] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[7] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[8] D. M. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, 2000.

[9] D. Burger and T. M. Austin. The SimpleScalar tool set. *SIGARCH Computer Architecture News*, 25(3):13–25, 1997. http://www.simplescalar.com.

[10] M. L. C. Cabeza, M. I. G. Clemente, and M. L. Rubio. CacheSim: A cache simulator for teaching memory hierarchy behaviour. In *ITiCSE '99: Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*, page 181, 1999.

[11] R. Carl and J. E. Smith. Modeling superscalar processors via statistical simulation. In *Workshop on Performance Analysis and its Impact on Design (PAID), held in conjunction with ISCA*, 1998.

[12] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, 2005.

[13] X. E. Chen and T. M. Aamodt. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 59–70, 2008.

[14] D. Chiou, H. Sanjeliwala, D. Sunwoo, J. Z. Xu, and N. A. Patil. FPGA-based fast, cycle-accurate, full-system simulators. In *Workshop on Architecture Research using FPGA Platforms, held in conjunction with HPCA-12*, 2006.

[15] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycle-accurate simulators. In *MICRO '07: Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, pages 249–261, 2007.

[16] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 76–87, 2004.

[17] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 468–477, 1996.

[18] J. Edler and M. D. Hill. Dinero IV: Trace-driven uniprocessor cache simulator. Available at: http://www.cs.wisc.edu/∼markhill/DineroIV/, 1999.

[19] L. Eeckhout, R. H. Bell Jr., B. Stougie, K. De Bosschere, and L. K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 350–361, 2004.

[20] L. Eeckhout and K. De Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 25–34, 2001.

[21] L. Eeckhout, K. De Bosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *ISPASS '00: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 1–6, 2000.

[22] L. Eeckhout and D. Genbrugge. *Processor, Multi-Core and System-on-Chip Simulation*, chapter Statistical Simulation. Springer, 2010.

[23] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. De Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, 2003.

[24] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *Computer*, 35(2):68–76, 2002.

[25] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.

[26] S. Eyerman, L. Eeckhout, T. S. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems*, 27(2):1–37, 2009.

[27] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *ISCA '94: Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 211–222, 1994.

[28] D. Genbrugge and L. Eeckhout. Statistical simulation of chip multiprocessors running multi-program workloads. In *ICCD '07: Proceedings of the 25th International Conference on Computer Design*, pages 464–471, 2007.

[29] D. Genbrugge and L. Eeckhout. Memory data flow modeling in statistical simulation for the efficient exploration of microprocessor design spaces. *IEEE Transactions on Computers*, 57(1):41–54, 2008.

[30] D. Genbrugge and L. Eeckhout. Chip multiprocessor design space exploration through statistical simulation. *IEEE Transactions on Computers*, 58(12):1668–1681, 2009.

[31] D. Genbrugge, L. Eeckhout, and K. De Bosschere. Accurate memory data flow modeling in statistical simulation. In *ICS '06: Proceedings of the 20th Annual International Conference on Supercomputing*, pages 87–96, 2006.

[32] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *HPCA '10: Proceedings of the 16th IEEE Symposium on High-Performance Computer Architecture*, 2010. Accepted for publication.

[33] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 102–113, 2004.

[34] A. Hartstein and T. R. Puzak. The optimum pipeline depth for a microprocessor. In *ISCA '02: Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 7–13, 2002.

[35] J. L. Henning. SPEC CPU suite growth: an historical perspective. *SIGARCH Computer Architecture News*, 35(1):65–68, 2007.

[36] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[37] C. Hsieh and M. Pedram. Microprocessor power estimation using profile-driven program synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(11):1080–1089, Nov. 1998.

[38] C. Hughes and T. Li. Accelerating multi-core processor design space evaluation using automatic multi-threaded workload synthesis. In *IISWC '08: Proceedings of the 4th IEEE International Symposium on Workload Characterization*, pages 163–172, 2008.

[39] E. İpek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS '06: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 195–206, 2006.

[40] V. S. Iyengar and L. H. Trevillyan. Evaluation and generation of reduced traces for benchmarks. Technical Report RC 20610, IBM Research Division, T. J. Watson Research Center, Oct 1996.

[41] V. S. Iyengar, L. H. Trevillyan, and P. Bose. Representative traces for processor models with infinite cache. In *HPCA '96: Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, pages 62–73, 1996.

[42] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 208–219, 2008.

[43] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[44] A. Joshi, L. Eeckhout, R. H. Bell, Jr., and L. K. John. Distilling the essence of proprietary workloads into miniature benchmarks. *ACM Transactions on Architecture and Code Optimization*, 5(2):1–33, 2008.

[45] T. S. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Workshop on Memory Performance Issues (WMPI), held in conjunction with ISCA*, 2002.

[46] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 338–349, 2004.

[47] T. S. Karkhanis and J. E. Smith. Automated design of application specific superscalar processors: an analytical approach. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 402–411, 2007.

[48] T. Kgil, S. D'Souza, A. Saidi, N. Binkert, R. Dreslinski, T. Mudge, S. Reinhardt, and K. Flautner. Picoserver: Using 3D stacking technology to enable a compact energy efficient chip multiprocessor. In *ASPLOS '06: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, 2006.

[49] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA '81: Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, 1981.

[50] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS '06: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 185–194, 2006.

[51] B. C. Lee, J. Collins, H. Wang, and D. M. Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 270–281, 2008.

[52] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS '01: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 164–171, 2001.

[53] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full-system simulation platform. *Computer*, 35(2):50–58, 2002.

[54] P. Michaud, A. Seznec, and S. Jourdan. An exploration of instruction fetch requirement in out-of-order superscalar processors. *Int. J. Parallel Program.*, 29(1):35–58, 2001.

[55] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *HPCA '06: Proceed-*

ings of the 12th IEEE Symposium on High-Performance Computer Architecture, pages 254–265, 2006.

[56] S. S. Mukherjee, S. V. Adve, T. M. Austin, J. Emer, and P. S. Magnusson. Performance simulation tools. *Computer*, 35(2):38–39, 2002.

[57] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. Huss-Lederman, and J. R. Larus. Wisconsin Wind Tunnel II: A fast, portable parallel architecture simulator. *IEEE Concurrency*, 8(4):12–20, 2000.

[58] D. B. Noonburg and J. P. Shen. A framework for statistical modeling of superscalar processor performance. In *HPCA '97: Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, pages 298–309, 1997.

[59] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 15–24, 2001.

[60] S. Nussbaum and J. E. Smith. Statistical simulation of symmetric multiprocessor systems. In *SS '02: Proceedings of the 35th Annual Simulation Symposium*, pages 89–97, 2002.

[61] M. Oskin, F. T. Chong, and M. Farrens. Hls: combining statistical and symbolic simulation to guide microprocessor designs. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 71–82, 2000.

[62] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer. Quick performance models quickly: Closely-coupled partitioned simulation on FPGAs. In *ISPASS '08: Proceedings of the IEEE International Symmposium on Performance Analysis of Systems and Software*, pages 1–10, 2008.

[63] D.A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D.I. August, and D. Connors. Exploiting parallelism and structure to accelerate the simulation of chip multi-processors. In *HPCA '06: Proceedings of the 12th IEEE Symposium on High-Performance Computer Architecture*, pages 29–40, 2006.

[64] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 244–256, 2003.

[65] P. Petoumenos, G. Keramidas, H. Zeffer, S. Kaxiras, and E. Hagersten. STATSHARE: A statistical model for managing cache sharing via decay. In *Workshop on Modeling, Benchmarking and Simulation (MoBS), held in conjunction with ISCA*, 2006.

[66] M. Rosenblum and M. Varadarajan. Simos: A fast operating system simulation environment. Technical report, Stanford University, Stanford, CA, USA, 1994. http://simos.stanford.edu.

[67] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS '02: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.

[68] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS '00: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[69] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic evaluation of shared-memory systems with ilp processors. In *ISCA '98: Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 380–391, 1998.

[70] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 24–35, 1993.

[71] T. M. Taha and S. Wills. An instruction throughput model of superscalar processors. *IEEE Transactions on Computers*, 57(3):389–403, 2008.

[72] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *MICRO 35: Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 271–282, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[73] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation. In *HiPEAC '05: Proceedings of the International Conference on High Performance Embedded Architectures and Compilation*, pages 47–67, 2005.

[74] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Considering all starting points for simultaneous multithreading simulation. In *ISPASS '06: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 143–153, 2006.

[75] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Representative multiprogram workloads for multithreaded processor simulation. In *IISWC '07: Proceedings of the 10th IEEE International Symposium on Workload Characterization*, pages 193–203, 2007.

[76] M. Van Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *ISPASS '04: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 45–56, 2004.

[77] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic. RAMP: Research accelerator for multiple processors. *IEEE Micro*, 27(2):46–57, 2007.

[78] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Simulation sampling with live-points. In *ISPASS '06: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 2–12, 2006.

[79] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 84–97, 2003.

[80] J. J. Yi, H. Vandierendonck, L. Eeckhout, and D. J. Lilja. The exigency of benchmark and compiler drift: designing tomorrow's processors with yesterday's tools. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 75–86, 2006.

[81] M.T. Yourst. PTLSim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS '07: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 23–34, 2007.