Efficiënte exploratie van de ontwerpruimte
van ingebedde microprocessors

Efficient Design Space Exploration of Embedded Microprocessors

Maximilien Breughe

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

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. R. Van de Walle
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2014 - 2015

UNIVERSITEIT
GENT

ISBN 978-90-8578-747-1
NUR 980, 950
Wettelijk depot: D/2014/10.500/93

Made in Texas.

Dankwoord

Dat ik na vijf jaar ijverig experimenteel onderzoek in mijn doctoraal examen aan de universiteit Gent geslaagd ben, stemt mij fier en gelukkig. Maar evenzeer realiseer ik mij dat dit succes de resultante is van een reeks ontmoetingen met wetenschappers van topniveau. En die mensen wil ik hier oprecht bedanken voor wat zij voor mij betekend hebben.

Vooreerst mijn twee promotoren: prof. Lieven Eeckhout en dr. Stijn Eyerman. Aan Lieven heb ik te danken dat ik een doctorale graad bekomen heb. Hij heeft mijn carrière vorm gegeven. Hij is mij onder de studenten lichting 2009 komen opzoeken met de vraag om in zijn dienst experimenteel werk te verrichten op het gebied van computerarchitectuur. Hij heeft mij gevormd in zijn lab en mij geholpen bij het uitwerken van mijn projecten. Hij heeft mij geleerd hoe publicaties op te stellen en me aangezet tussendoor op internationale topconferenties en in internationale tijdschriften vier artikelen te publiceren. Hij was het die mij in de juiste richting duwde en mij in het buitenland stage liet lopen. Ik ben trots een leerling geweest te zijn van deze professor met internationale faam.

Aan Stijn, mijn tweede promotor, eveneens mijn dank. Die heeft mij zeer intens begeleid bij mijn experimenten. Met zijn ongeëvenaarde expertise en scherp inzicht heeft mij telkens weer de juiste richting doen inslaan. Ook hij was enthousiast over de mogelijkheden die mij in Amerika ter beschikking werden gesteld. Samen met Lieven heeft hij ervoor gezorgd dat mijn onderzoek in Gent en Austin vloeiend verliepen en voor deze harmonieuze samenwerking wil ik ze allebei nog eens extra bedanken.

De tweede groep wetenschappers die mij geleid hebben zijn mijn Amerikaanse medewerkers. *I hereby would like to thank Nasr Ullah, Wichaya Top Changwatchai, Brian Grayson and Tim Snyder for making this internship happen. These four persons were key in introducing me into Samsung Austin R&D Center and I don't have enough words to express my thanks. The internship has been invaluable for my PhD. A special word of thanks goes to David Eklöv, for sharing his experience as a PhD student and for his willingness to listen to my research topic. Another special word of thanks goes to Shen, for her patience, support, wise advice and motivating words, especially during the months I wrote the dissertation. I would also like to thank Kshitij and Zheng Li for their wise practical advice.*

Tot nu toe heb ik de wetenschappers bedankt die mij in de opbouw van mijn werk geholpen hebben. Ik wil echter evenzeer mijn dank richten tot diegenen die de zware opdracht aanvaard hebben in de examencommissie te zetelen. Deze bestond, naast Lieven en Stijn, uit zes leden, zijnde vier binnenlandse leden, nl. prof. Koen De Bosschere, prof. Filip De Turck, dr. Philippe Manet en prof. Luc Taerwe, en twee buitenlandse professoren, nl. Prof. Roy Jenevein en Prof. Erik Hagersten. Ik bedank ze allen één voor één voor het extra werk dat zij op zich genomen hebben met mijn scriptie door te lezen, wat suggesties naar bijschaving naar voren te brengen en uiteindelijk een oordeel te vellen. En voor de twee buitenlandse professoren komt daar nog de zware trip naar Gent bij. *Many thanks go to Prof. Erik Hagersten and Prof. Roy Jenevein for their effort to evaluate this thesis, their suggestions and their willingness to travel to Ghent, despite their busy schedules.*

Verder wil ik enkele zaken uit de persoonlijke sfeer niet onvermeld laten. Vooreerst, de sfeer van het thuisfront zal ik nooit vergeten. Kristof en Klaas, mijn beide "bureaugenoten" tijdens mijn doctoraatsjaren, hebben mij, naast wetenschappelijk advies, veel morele steun geboden en voor de nodige afleiding gezorgd. Ook de overige bureaugenoten, nl. Cecilia, Sam, Sander en Shoaib hebben meegeholpen aan het creëren van een aangename stimulerende werksfeer. Daarnaast wil ik niet vergeten, omwille van hun professionele ondersteuning, Marnix, Michiel, Ronny en Pieterjan evenals het personeel van het Vlaams Supercomputer Center (VSC). Vervolgens – zo hoort het – een woordje van dank aan mijn familie. Daarbij denk ik in het bijzonder aan: mama, papa en bonpa die mij de kans gegeven hebben om te geraken waar ik nu sta; en mijn zus Laétitia die er altijd de vrolijke noot wist in te houden. Wie ik echter tot slot nog in herinnering wil brengen is mijn studiegenoot en vriend Geert, die op het moment van schrijven voor een bekende Belgische firma ergens in Australië aan het baggeren is.

Maximilien Breughe
Gent, 25 november 2014

Examencommissie

Prof. Luc Taerwe, *voorzitter*
Prodecaan Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Koen De Bosschere, *secretaris*
Vakgroep ELIS
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

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

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

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

Prof. Erik Hagersten
Uppsala University
Sweden

Prof. Roy Jenevein
The University of Texas at Austin
USA

Dr. Philippe Manet
Embedded Computing Specialists
Brussel

Leescommissie

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

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

Prof. Erik Hagersten
Uppsala University
Sweden

Prof. Roy Jenevein
The University of Texas at Austin
USA

Dr. Philippe Manet
Embedded Computing Specialists
Brussel

Samenvatting

Ondanks hun sterke vertegenwoordiging in het alledaagse leven is het ontwerpen van microprocessors bijzonder complex. Het vereist het werk van honderden mensen gedurende enkele jaren. Dit komt door de strenge eisen die worden opgelegd aan het ontwerp: processors moeten snel zijn om de steeds complexere software te kunnen uitvoeren, maar ook vermogenefficiënt om een lange batterijduur te garanderen. Om deze criteria te kunnen bereiken kunnen we gebruik maken van de revolutionaire vooruitgang op gebied van technologie. Dit laat ons toe om miljarden transistors te plaatsen op een kleine oppervlakte. Deze transistors bouwen de verschillende componenten van de microprocessor op. De vele componenten waaruit de microprocessor bestaat hebben verschillende parameters die moeten worden gekozen om ontwerpcriteria zoals uitvoeringstijd en vermogen-efficiëntie te bereiken. Het vinden van de juiste parameters voor iedere component om deze ontwerpcriteria te bereiken is het onderwerp van ontwerpruimte-exploratie (Eng.: design space exploration).

Traditionele cyclus-getrouwe simulators zijn niet geschikt om grote ontwerpruimtes volledig te exploreren door de lage simulatiesnelheid. Trage simulaties leiden namelijk tot twee grote problemen. Ten eerste worden computerarchitecten verplicht om slechts enkele gebieden in de grote ontwerpruimte te simuleren. Hierdoor bestaat de kans dat interessante alternatieve ontwerpen nooit gesimuleerd worden. Ten tweede limiteert de lage simulatiesnelheid de werklasten die gebruikt worden tijdens de exploratie van de ontwerpruimte. Dit kan ertoe leiden dat het gekozen ontwerp suboptimaal zal presteren voor werklasten die niet tijdens het ontwikkelingsprocess gebruikt werden.

Om computerarchitecten te helpen bij de selectie van de gebieden die ze wensen te simuleren, introduceren we een snel analytisch prestatiemodel dat voorafgaand aan de cyclus-getrouwe simulaties gebruikt kan worden. De constructie van het model is gebaseerd op de interne werking van de microprocessors. We valideren de nauwkeurigheid zowel ten opzichte van cyclus-getrouwe simulatie als ten opzichte van echte hardware. Door het model te vergelijken ten opzichte van meer dan 6,000 cyclus-getrouwe simulaties (een combinatie van programma-applicaties en microarchitecturale configuraties), tonen we aan dat het model het aantal cycli per in-

structie (CPI) kan voorspellen met een gemiddelde absolute voorspellingsfout van slechts 2.8% en een standaardafwijking van 0.024. Wanneer we het model inzetten om de prestatie te voorspellen van 19 applicaties voor ingebedde systemen op de ARM Cortex-A8 op een hardware-platform, tonen we aan dat het een gemiddelde absolute nauwkeurigheidsfout in CPI heeft van slechts 10%, met een standaardafwijking van 0.10. Het eigenlijke model bestaat uit een som van verschillende termen die de uitvoeringstijd op een verschillende manier beïnvloeden, zoals bijvoorbeeld foutief voorspelde sprongen, cache missers, afhankelijkheden tussen instructies, maar ook het onbeschikbaar zijn van functionele eenheden. De invoer voor het model bestaat uit de configuratie van de hardware waarvoor een prestatieschatting gewenst is, een aantal programmakarakteristieken die onafhankelijk zijn van de microarchitectuur, en programmakarakteristieken die gedeeltelijk afhankelijk zijn van de microarchitectuur. Het voordeel om de totale prestatie onder te verdelen in aparte termen is dat op die manier de interactie van een programma met de microarchitectuur gevisualiseerd kan worden. We gebruiken het model voor het oplossen van een aantal ontwerpproblemen. We tonen aan dat we het model kunnen inzetten om het aantal functionele eenheden te minimaliseren voor een gegeven prestatiedoel. Verder zetten we het model in om de microarchitectuur te optimaliseren voor zowel prestatie als energie. Naast deze ontwerpproblemen geven we ook een aantal interessante inzichten mee door de interactie tussen programma's en de microarchitectuur te visualiseren aan de hand van het model. We tonen aan dat programma's met een gelijke instructie-mix sterk verschillend kunnen reageren op het schalen van microarchitecturale componenten. Wanneer we de interactie tussen een programma en de microarchitectuur visualiseren voor uitvoerbare bestanden gegenereerd met verschillende compiler-instellingen, kunnen we aantonen hoe deze instellingen de prestatie beïnvloeden.

Het kiezen van de juiste werklasten die tijdens het ontwerp van microprocessors gebruikt worden is van groot belang om robuste beslissingen te maken en te vermijden dat het ontwerp slecht presteert op nieuwe werklasten. Tot op vandaag hebben voorafgaande studies vooral gefocust op het vinden van representatieve programma's en representatieve regio's binnen deze programma's. Er zijn echter twee impliciete parameters die gekoppeld gaan met de gekozen werklast die slechts weinig bestudeerd zijn: de programma-invoer en de compileroptimalisatievlaggen die gebruikt werden om een uitvoerbaar bestand te genereren. Met een grootschalig onderzoek tonen we aan dat slecht gekozen programma-invoerbestanden kunnen leiden tot een ontwerp met een energie-efficiëntie dat 57% lager ligt dan het ontwerp met de hoogste energie-efficiëntie. Een slecht gekozen uitvoerbaar bestand kan resulteren in een energie-efficiëntie dat 16% lager ligt dan het ontwerp met de hoogste energie-efficiëntie.

Gezien de potentieel sterk negatieve impact van slecht gekozen pro-

gramma-invoerbestanden introduceren we drie verschillende karakterisatietechnieken om de kans op een nadelige energie-efficiëntie in te perken. Deze technieken hebben als grote voordeel dat, eens de invoerbestanden gekarakteriseerd zijn, de simulatietijd niet stijgt ten opzichte van willekeurig geselecteerde invoerbestanden: onze experimenten tonen aan dat ten hoogste drie representatieve invoerbestanden voldoende zijn om een ontwerp te vinden dat nagenoeg de hoogste energie-efficiëntie heeft. De eerste techniek maakt gebruik van kleine regio's in de ontwerpruimte om de invoerbestanden te filteren, wat tot een energie-efficiëntie leidt die slechts 7% onder het maximum ligt wanneer drie invoerbestanden gebruikt worden. Deze techniek vereist een aantal simulaties, evenredig aan de grootte van de regio's en het beschikbaar aantal invoerbestanden, voorafgaand aan de eigenlijke ontwerpruimte-exploratie om de invoerbestanden te karakteriseren. De tweede techniek karakteriseert de invoerbestanden onafhankelijk van de onderliggende microarchitectuur, op basis van basic-block-vectoren (BBV's). Deze techniek heeft de kortste karakterisatietijd en leidt tot een energie-efficiëntie die slechts 4% onder het maximum ligt wanneer twee invoerbestanden gebruikt worden. De derde en laatste techniek maakt het mogelijk om met drie invoerbestanden het ontwerp te vinden met de hoogste energie-efficiëntie. Hiervoor maakt de techniek gebruik van simulatiestatistieken zoals het gemiddeld aantal cycli per instructie (CPI). De karakterisatietijd komt overeen met een enkele simulatie voor ieder invoerbestand.

Als besluit geloven we dat deze twee bijdragen, het analytisch prestatiemodel en de karakterisatietechnieken voor programma-invoer, een belangrijk deel moeten vormen van de ontwerpcyclus voor microprocessors. Het prestatiemodel laat ons toe om prestatieschattingen te maken in enkele seconden, in plaats van uren cyclus-getrouwe simulatietijd. Deze snelle prestatieschattingen zijn van groot belang om interessante regio's te ontdekken in de grote ontwerpruimte. Met de karakterisatietechnieken voor programma-invoer kunnen we representatieve programma-invoerbestanden selecteren tijdens het ontwerp van de microprocessor. Dit zorgt ervoor dat het uiteindelijke ontwerp voldoende robuust is voor het uiteenlopende dynamisch gedrag van computerprogramma's.

Summary

Despite their ubiquitous presence in everyday life, designing a microprocessor is a complicated process, involving the work of hundreds of people for several years. Not only should new microprocessors keep up with the performance demands of new software applications, they also need to be power efficient. To achieve these design goals we can rely on spectacular advances in technology that provide the ability of placing billions of transistors on a very small area. These transistors are hierarchically organized into a number of components that all require the appropriate scaling and tuning to meet the design goals. The large number of these components together with the many parameters they bring along build up a large design space for the computer architect to explore.

Traditional cycle-level simulators are not suitable to explore these large design spaces because of their low simulation speed. The low simulation speed introduces two important problems. First, slow simulation forces computer architects to carefully select small regions in the design space they wish to simulate, potentially leaving out interesting design alternatives. Second, the low simulation speed further limits the amount and sizes of workloads that are used during design space exploration, possibly resulting in designs that are suboptimal for workloads that were not simulated during the development.

To guide architects in selecting interesting regions in the typically large design spaces, we propose a fast analytical performance model that can be used in the early stages of the design. We construct the model based of the internal mechanics of the microarchitecture, and compare it against both detailed cycle-level simulation and hardware. For over 6,000 cycle-level simulations (a combination of workloads and microarchitectural configurations), we show that the model's predicted number of cycles per instructions (CPI) has an absolute error of 2.8% on average, with a standard deviation of 0.024. For a set of 19 embedded benchmarks, executed on a hardware platform with the ARM Cortex-A8 processor, we report an absolute average prediction error in CPI of 10% on average, with a standard deviation of 0.10. The final model consists of the sum of a number of penalty terms that reflect the impact of miss events (i.e., branch mispredictions, cache misses, etc.), inter-instruction dependences and functional

unit contention. The inputs to provide to the model are a number of machine parameters for which a performance estimate is desired, a number of program characteristics independent on the microarchitecture, and a number of mixed program-machine characteristics. The advantage of having separate terms that reflect a penalty is that they can be used to visualize the interaction of an application with the microarchitecture. We use the model for a number of design space exploration studies: we are able to find the minimum number of functional units to achieve a predefined performance target using the model, as well as to find the microarchitecture with an energy-delay-product within 1% of the microarchitecure with the lowest energy-delay-product as found by detailed cycle-level simulation. We further reveal a number of interesting insights by using the model to visualize the application-microarchitecture interaction. We show that applications with a similar instruction mix can react very differently on microarchitectural enhancements. By visualizing the application-microarchitecture interaction on a number of different optimized binaries for the same application, we show how compiler optimization flags can impact the performance of an application.

Selecting representative workloads to use throughout design space exploration is of extreme importance in order to assure that design decisions are robust across previously unseen workloads. Before, workload selection has mainly focused on finding representative applications and finding representative samples within these applications. There are however two implicit parameters tied to the workload that have been given only little attention until now: the application's input data sets and the compiler optimization flags used to generate the application binary. We show that poorly chosen application inputs could guide design space exploration to a design with an energy-delay-product (EDP) 57% higher than the design with the lowest EDP. A poorly chosen application binary could result in an EDP increase of 16% over the design with the lowest EDP.

Given the potentially high impact of poorly chosen application inputs we introduce three input selection techniques to reduce the EDP deficiency, without drastically increasing the simulation time: in our experimental setup, we find one to three inputs to be sufficient to find a nearly optimal design. Filtering the inputs by performing design space exploration on a very small region of the original design space reduces the worst case scenario to an EDP deficiency of 7% by using three inputs. This technique requires a number of simulations, proportional to the number of inputs available and the size of the selected region, prior to design space exploration to profile the inputs. With basic block vector (BBV) selection we characterize inputs in a microarchitecture-independent way. This technique has the lowest input characterization overhead and is able to reduce the worst case scenario to an EDP deficiency of 4% with as few as two inputs. The third technique is able to completely eliminate the EDP deficiency in our

setup by using three inputs during design space exploration. It therefore characterizes inputs by using simulation statistics, such as CPI (cycles per instruction), from a single cycle-level simulation per input.

We believe that the two main contributions of this dissertation, namely the analytical performance model and benchmark input selection techniques, are key additions into the microarchitectural design cycle. With the proposed mechanistic model, we are able to estimate microprocessor performance in seconds, compared to hours of cycle-level simulation, which helps guiding us into interesting regions of the typically large design space. The proposed input selection techniques allow us to use representative benchmark inputs during design space exploration, leading to design decisions that are optimal across inputs.

Contents

List of Tables

List of Figures

List of Abbreviations

ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
BBV	Basic Block Vector
CISC	Complex Instruction Set Computer
CPI	Cycles Per Instruction
DV	Design Verification
EDP	Energy-Delay Product
EPI	Energy Per Instruction
FDTD	Finite Difference Time Domain
GCC	GNU Compiler Collection
ILP	Instruction-Level parallelism
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
L1 I-cache	Level one Instrution cache
L1 D-cache	Level one Data cache
L2 cache	Level two cache
LSU	Load/Store Unit
MLP	Memory-Level Parallelism
MSHR	Miss Status Handling Register
RAW	Read After Write
RTL	Register Transfer Level
SHA	Secure Hasing Algorithm
TLB	Translation Lookaside Buffer
TPI	Time Per Instruction
WAR	Write After Read
WAW	Write After Write

Chapter 1

Introduction

I think there is a world market for maybe five computers.
Thomas J. Watson, IBM

Microprocessors take an important role in our everyday life: several billions of microprocessors are distributed in the world in many different devices [1], ranging from smartphones and smart watches to desktop computers, game consoles and server systems in data centers. Despite their ubiquitous presence, building a microprocessor is a very complicated process. Not only does the microprocessor need to keep up with the increasing complexity of software applications, it also needs to be power efficient, both to allow for a long battery lifetime in handheld devices and to keep the energy bill low and cooling costs reasonable in big data centers. To achieve these goals we can rely on spectacular advances in technology that provide the ability of placing billions of transistors on a very small area. The task of organizing these transistors, however, is a very complicated one and involves the work of hundreds of people for several years [12].

Figure 1.1 shows a high-level view of the main steps during the development of a microprocessor. During every step of the development, performance, power/energy consumption, area, reliability, etc., are the optimization criteria. It is the task of the computer architect to build an RTL design that meets these criteria. Once the RTL design is ready (and functionally verified by the Design Verification teams (DV)), it will go through the physical design steps to create a layout file of the circuit (GDSII-file), which is sent to the fab for wafer manufacturing.

To understand the impact of design decisions, the RTL design can be simulated in software, which gives a very accurate estimate of its performance if it were to be built in silicon. However, there are two main disadvantages. First, only very small workloads can be simulated because of the low simulation speed: typically, simulating an application on RTL is 7 to 8 orders of magnitude slower than executing an application on real

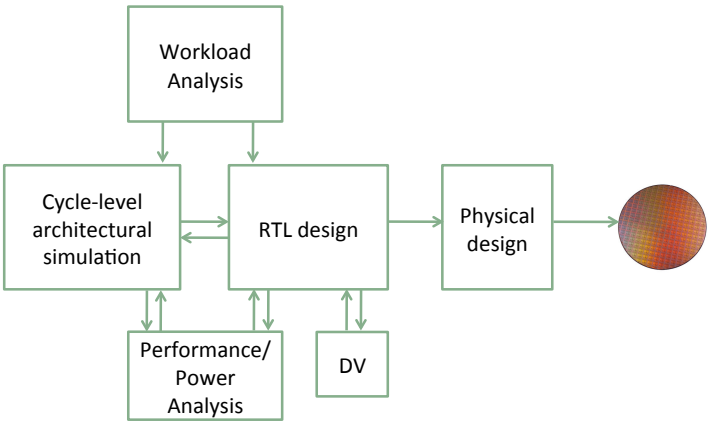


Figure 1.1: Schematic view of the main steps involved in microprocessor design.

hardware. To make design decisions, however, it is important to be able to estimate the performance of workloads that are sufficiently large. Second, RTL code tends to be inflexible when trying to explore the behavior of new design features. To overcome these disadvantages, cycle-level microarchitectural simulators are built and used by performance and power analysis teams to help RTL designers make design decisions. Cycle-level architectural simulators aim at making performance estimates that are close to those of RTL simulators, but they operate at a higher level of abstraction. Because of this they are 3 to 4 orders of magnitude faster than RTL simulators, and hence more design configurations can be explored and more and larger workloads can be simulated. Unfortunately, these architectural simulators are still up to 4 orders of magnitude slower than execution on real hardware, which forms a hard limit on the number of simulations that can be done. This potentially leads to designs that are suboptimal for workloads that were not simulated during the development and/or limits the number of designs that computer architects can simulate.

1.1 Motivation and focus

Traditional cycle-level simulators are clearly not suitable to explore large design spaces. Because resources are limited, designers are tied to a limited simulation budget and hence are forced to carefully select small regions in the design space they wish to simulate. A second problem is that the low simulation speed further limits the amount and sizes of workloads to be simulated. However, in order to ensure that design decisions are robust, it is important to select the appropriate workloads during design space exploration. Figure 1.2 shows our contributions to the microprocessor development cycle to help solving both these problems.

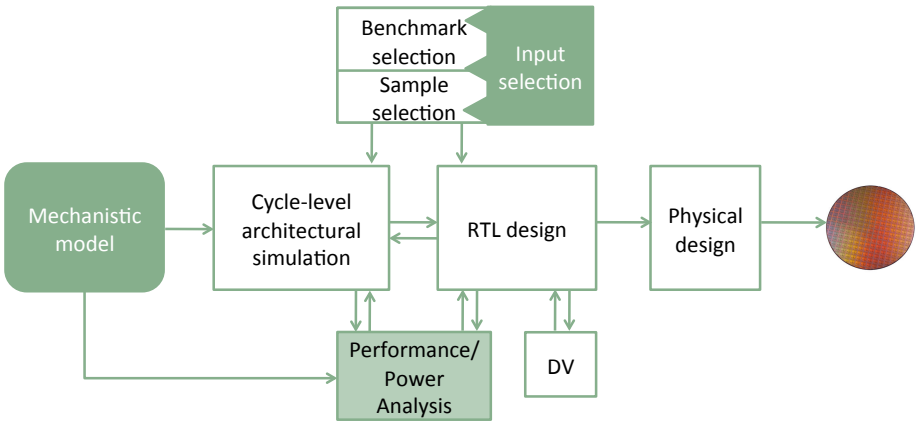


Figure 1.2: The contributions of this thesis (mechanistic model and input selection) and how they affect processor design.

We develop an analytical performance model, built from the internal mechanics of a microprocessor, for exploring design spaces in the early design stage, by trading off accuracy for speed. Not only does the model speed up simulation time drastically as it involves evaluating a number of mathematical formulas, it also provides insight into the application-microarchitecture interaction. The main intent of the model is to get quick insight and to prune down the large number of designs to a smaller set that can be simulated with detailed cycle-level simulation.

The challenge in workload selection is to select as few workloads as possible while capturing as much program-behavior as possible. Limiting the number of workloads is important to reduce simulation time, while having a broad coverage reduces the chance that our design has an undesired performance on unseen workloads once the microprocessor is shipped. We contribute to the workload selection process by adding techniques for benchmark input selection to the computer architect's toolbox.

When moving towards application-specific processors, the contributions of this thesis gain even more importance. An application-specific processor is a processor that is tailored for the application or application domain of interest. This specialized processor is optimized to achieve the best possible performance within a given energy envelope, or, vice versa, the processor is optimized to consume the least possible energy while achieving a given performance target [1] . Application-specific processors are more cost-effective than hardware specialization, such as application-specific integrated circuits (ASICs), and in addition benefit from the programmability of a general-purpose processor. In order to find the sweet spot in the typi-

[1]Synopsys' DesignWare ARC processors and Cadence Tensilica Xtensa dataplane processors are solutions along this line.

cally huge design spaces, optimized along multiple design criteria, the design methodology is obviously key. The analytical performance model that we introduce in this thesis can help computer architects guide design space exploration by creating fast performance predictions for large regions in the design space. Further, since these specialized processors are customized for a specific application it is important to capture sufficient dynamic behavior of the application. Therefore, the input selection techniques that we propose allow computer architects to find inputs that are representative for the application.

As stated earlier, a primary design goal for embedded microprocessors is energy-efficiency. While superscalar out-of-order processors deliver high performance, they come at the price of complex logic which makes them power-hungry. Because the performance of a single core is proportional to the square root of its area [2] [5], known as Pollack's rule, and power consumption is roughly proportional to the area, processors that occupy a smaller area are more power-efficient. This is even further motivated by the end of the Dennard scaling [13] where the power density increases when transistor sizes further decrease. We therefore consider superscalar in-order processors throughout the course of this thesis, as they are less complex to design, occupy less area and consume less energy, compared to superscalar out-of-order processors. In-order processors are widely used in today's embedded systems such as in the iPhone 4 and iPad (under the Apple A4 processor), but also in the recently released Galaxy S5 Mini (using the ARM Cortex-A8) and the high-end Samsung Galaxy S5 smartphone (as part of the big.LITTLE core [27]).

1.2 The Contributions of this Thesis

1.2.1 Contribution 1: Fast microarchitectual evaluation and bottleneck visualization

The intent of cycle-level microarchitectural simulators is to allow computer architects to make high level decisions of a complicated hardware design. Computer architects build these microarchitectural simulators to be as accurate as possible compared to RTL-simulators or real hardware. Because cycle-level microarchitectural simulators have a higher level of abstraction than RTL-simulators they are 3 to 4 orders of magnitude faster. Therefore computer architects can use them to explore many different potential microarchitectural designs. Unfortunately, applications executed on a simulator are still up to 4 orders of magnitude slower compared to executing

[2] The performance of a single core is proportional to the square root of its complexity. Hence, when considering the same feature size, the performance of a single core is proportional to the square root of its area.

these applications on real hardware. Hence, an exhaustive search of all potential microarchitectural designs is impossible, forcing computer architects to carefully select the designs they want to simulate.

To guide architects selecting interesting regions in the typically huge design spaces we extend current practice of design space exploration by constructing a model with a higher level of abstraction. We build a performance model for superscalar in-order processors that eliminates the slowdown from which cycle-level simulation suffers by using analytical formulas.

As a bonus, the model not only reduces simulation time significantly, it also shows how an application interacts with the microarchitecture. This allows computer architects to visualize where an application's cycles are spent. In addition, when applying this visualization technique on multiple binaries of the same application, we are able to visualize how compiler flags impact the interaction between an application and the microarchitecture.

Our results show that the model has an absolute error of only 2.8% on average when compared against cycle-level simulation and 10% when compared against hardware. The evaluation time of the model is less than a second, while the evaluation time of cycle-level simulators typically takes several hours.

A discussion of the model has been peer reviewed by and presented at the 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), where it was **nominated for best paper award**:

> Maximilien Breughe, Stijn Eyerman, and Lieven Eeckhout, "A mechanistic performance model for superscalar in-order processors", ISPASS '12: *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software*.

An extended version of the model is in press for ACM Transactions on Architecture and Code Optimization (TACO):

> Maximilien Breughe, Stijn Eyerman, and Lieven Eeckhout, "Mechanistic analytical modeling of superscalar in-order processor performance", TACO '14: *ACM Transactions on Architecture and Code Optimization*.

1.2.2 Contribution 2: Selection methodology for representative benchmark inputs

When performing design space exploration there are two parameters that are implicit to the workloads that are being used: the input to the application and the set of compiler flags that is used to optimize the application's

binary. Common practice is to use one or a couple of inputs and application binaries, and assume that they capture enough dynamic behavior to be representative for the entire application. These assumptions have never proven to be true and the need to study them is increasing when moving toward application-specific processors.

Therefore, we first conduct an experiment that uses a large database of inputs for a number of applications and show that some inputs could lead to poor results, when used to drive design space exploration. On average, the energy-delay product (EDP) [3] can be 57% or 33% higher than the lowest achievable EDP, when using one, respectively three badly chosen inputs. Second, we set up a similar experiment by using application binaries, compiled with different sets of compiler flags, to drive design space exploration. We show that the choice of compiler flags with which the application binary was generated, has a less significant impact: in the worst case the EDP is 16% higher on average than the lowest achievable EDP.

A discussion on the sensitivity study of benchmark inputs has been peer reviewed by and presented at the 2011 IEEE Symposium on Application Specific Processors (SASP):

> Maximilien Breughe, Zheng Li, Yang Chen, Stijn Eyerman, Olivier Temam, Chengyong Wu and Lieven Eeckhout. "How sensitive is processor customization to the workload's input datasets?", SASP '11: *Proceedings of the 2011 IEEE 9th Symposium on Application Specific Processors*.

Based on these insights we conclude that it is important to perform design space exploration with representative inputs. It is unclear, however, a priori which input is representative for the application. Hence, without a proper selection method we would either run the risk of building a design with diminishing performance by using non-representative inputs, or we would need to perform the exploration with a high number of inputs and increase simulation time drastically. Both options are infeasible in practice. Therefore we build different input selection techniques to reduce the EDP deficiency. With the techniques we propose we are able to reduce the average EDP deficiency to less than 3.7%, without a significant increase in simulation time.

A discussion of these input selection techniques and their trade-offs has been published in ACM Transactions on Architecture and Code Optimization (TACO) and has been presented at the 2014 International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC):

[3] We use EDP as a metric for energy-efficiency.

Maximilien Breughe and Lieven Eeckhout. "Selecting representative benchmark inputs for exploring microprocessor design spaces", TACO '13: *ACM Transactions on Architecture and Code Optimization, Vol.10, No.37, Dec. 2013.*

1.3 Thesis Outline

This dissertation is organized as follows. Chapter 2 describes the microarchitecture of modern superscalar in-order and out-of-order processors and describes the current state-of-the-art of analytical performance modeling and workload selection. Chapter 3 describes our analytical performance model for superscalar in-order processors and illustrates its usefulness with a number of case studies. In Chapter 4 we quantify the impact of implicit workload parameters on design space exploration and introduce our novel techniques to select representative benchmark inputs. Finally we present our conclusions in Chapter 5 and provide suggestions for future work.

Chapter 2

Background

An expert is a person who has made all the mistakes
that can be made in a very narrow field.
Niels Bohr

In this chapter we describe important background topics to situate the contributions of this work. We start by describing a high-level view of the architecture of superscalar in-order and out-of-order processors. Next we give an overview of prior work in analytical performance modeling of superscalar processors. The last section of this chapter describes the current state-of-the-art in workload selection.

2.1 Superscalar Processors

Microprocessors have been around for several decades, dating back from the 1970s [6]. Their purpose is to execute instructions, as defined by the programmer. Between the start and end of the execution of an instruction on a microprocessor there is a long and complex chain of logic gates. The longer the chain of logic gates, the longer the clock cycle of a single instruction and hence, the longer it takes to execute a software application.

Processing an instruction can be broken up into sequential pipeline stages in an *instruction pipeline*. The chains of logic gates in each of the sequential stages are therefore much smaller and hence the complexity of the work that needs to be done in each clock cycle is drastically reduced and spread over multiple stages. This means the length of the clock cycle can be reduced and instructions can theoretically flow into the pipeline cycle after cycle. Modern pipelines (such as Intel Core i7) use this principle to increase their clock frequency with deeply pipelined processors. The cost of placing buffers (among other logic) between the pipeline stages and the increasing performance penalty of branch mispredictions limit the amount

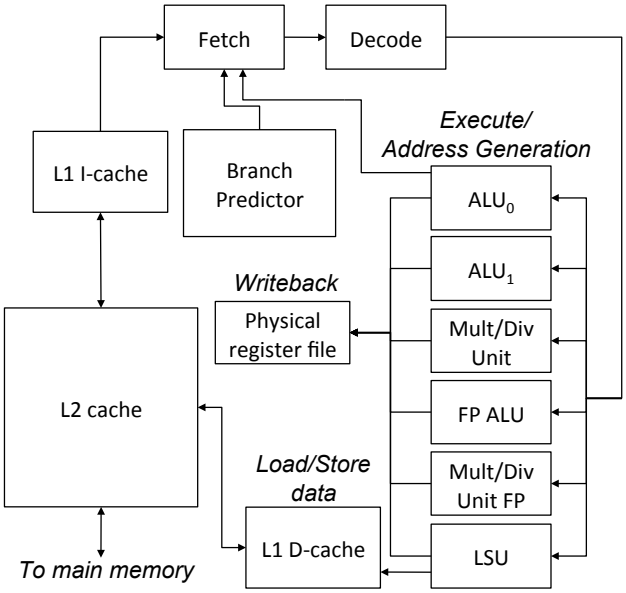


Figure 2.1: Block diagram of a superscalar in-order pipeline.

of pipelining.

A second step in increasing the rate at which instructions are executed is increasing the number of instructions that can be processed by a pipeline stage *concurrently*. Many applications have a certain amount of instruction-level parallelism (ILP): instructions that are independent of each other's results can be executed in parallel. Superscalar processors are built to exploit the instruction-level parallelism of applications. There are basically two kinds of superscalar processors: in-order processors and out-of-order processors. We will discuss both of them in the next subsections.

2.1.1 Superscalar in-order processors

Figure 2.1 shows the block diagram of a superscalar in-order pipeline. Instructions and data are accessed from the memory subsystem, and processed by the other units of the pipeline. The memory subsystem is designed to provide instructions and data to the processor as soon as possible by organizing the data in a hierarchical system. The execution of an instruction can be broken up into five stages: instruction fetch, instruction decode, execute, memory access and writeback. We consider a width W for each of the pipeline stages, meaning that a maximum of W instructions can flow into each pipeline stage every cycle. [1]

[1]Note that, to ease this discussion, we limit ourselves to a processor of depth 5 and a uniform width for each of the stages. The depth can be chosen arbitrarily and each pipeline

We will now describe the flow of instructions through the pipeline, stage by stage.

Instruction Fetch During instruction fetch, the processor fetches new instructions from the level-one instruction cache (L1 I-cache) of the memory subsystem. The fetch unit does this in a sequential way until a control flow instruction (i.e., a branch instruction) redirects the flow of instructions. The result of branch instructions, however, is only known during the execute stage, which means that the processor might fetch too many instructions on the sequential path. Because this has an important impact on performance, modern processors have branch predictors, predicting the outcome of a branch instruction early in the pipeline.

Instruction Decode After the instruction is fetched from the memory subsystem it will get decoded by the decode unit/stage which identifies the work that must be done. The decode unit/stage sets up the correct signals in order to process the instruction further: it determines which type of functional unit needs to be accessed based on the instruction type (addition, multiplication, load data, etc.) and which registers need to be accessed from the register file. In addition, depending on the instruction-set architecture (ISA) the processor implements, the decode unit/stage might need to determine the length of the instruction [2] , complicating the logic of the decode unit/stage even further.

For in-order processors, the decode unit/stage also handles the signalling for inter-instruction dependences. When one of the instructions that is decoded, depends on the outcome of an instruction that has not computed its result yet, the decode unit/stage blocks this instruction and all subsequent instructions from going to the next pipeline stage. An instruction in the decode unit/stage can be dependent on instructions in the execute, memory and writeback stages but also on older instructions in the decode unit/stage itself. To avoid waiting on producers to reach the *writeback* stage, forwarding logic is built between the pipeline stages. However, the pipeline will still be blocked until the result is *computed* in the execute stage. As we will see in Chapter 3, stalls due to inter-instruction dependences can be a limit on performance for a large number of applications. This, in addition to the associated hardware cost, limits the maximum width to benefit from instruction-level parallelism.

stage can potentially be designed with different widths.

[2] CISC processors (Complex Instruction Set Computer) have instructions of variable length.

Execute During execute, an instruction is processed by the appropriate functional unit. In Figure 2.1 we distinguish six functional units: two integer Arithmetic Logic Units (ALUs), an integer multiply/divide unit, a floating-point ALU, a floating-point multiply divide unit and a load/store unit (LSU). [3]

Integer ALU instructions, such as integer additions usually take a single clock cycle to execute. This means that each clock cycle a new ALU instruction can get processed by the same ALU. More complex instructions, such as floating-point operations can take multiple clock cycles before the result is calculated. We further refer to these instructions as long-latency instructions. Long-latency instructions not only take multiple cycles to compute their result, they also block the functional unit from processing new instructions for several cycles. To overcome this, functional units for long-latency instructions can have a pipelined design. This means that the computation of e.g., an integer multiply instruction, is further subdivided in smaller steps, allowing several multiply instructions to be in-flight in the same multiply unit. However, this complicates the design more and is not possible for all long-latency instructions.

The number of functional units is another important design parameter. When, for example, the processor pipeline has a width of $W = 4$, and only two ALUs are built in the processor, the processor will stall when more than two ALU instructions are coming out of the decode stage. This is called functional unit contention: the processor stalls because of an insufficient number of functional units. Depending on the target application domain, the number of each type of functional units must be carefully selected. Chapter 3 shows the optimal number of functional units of each type for a large number of applications of the SPEC CPU 2006 and MiBench benchmark suites.

Memory Load and store instructions use the LSU to access the memory subsystem in this stage. A load instructions first looks up whether the required data can be found in the fast level-one data cache (L1 D-cache). If the data cannot be loaded from the L1 D-cache, a request is sent out to the level-two cache (L2 cache). If the L2 cache does not have the data either, the request is sent further down the memory subsystem (e.g., L3 cache or directly to main memory).

Writeback During the writeback stage, computation results are written to the registers in the physical register file.

[3] Note that although the LSU unit is depicted under execute, it is accessed during the memory stage.

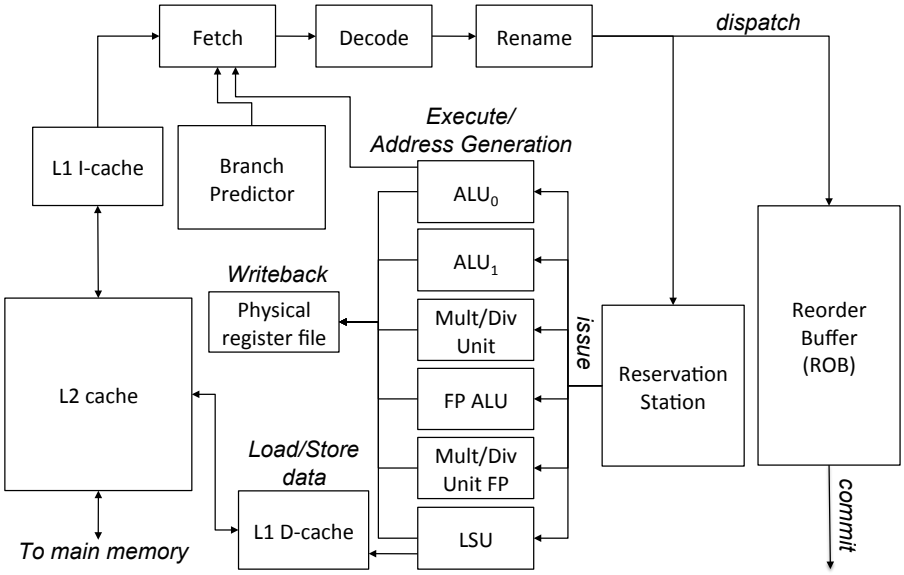


Figure 2.2: Block diagram of a superscalar out-of-order pipeline.

Examples of commercial superscalar in-order microarchitectures are ARM's Cortex-A7, ARM's Cortex-A8, Intel's Bonnell [4] , etc.

2.1.2 Superscalar out-of-order processors

Out-of-order processors remove the limit of executing instructions in program order. Instead of blocking the processor whenever an instruction waits for a dependence to resolve or a functional unit to become available, it looks whether it can execute subsequent instructions. Hence, out-of-order processors are capable of exploiting ILP even further. However, this requires additional logic: additional pipeline stages, a rename unit, a re-order buffer (ROB) and (a) reservation station(s). In addition, out-of-order processors improve memory-level parallelism (MLP) by adding miss status handling registers (MSHRs).

While this additional logic improves the ability to exploit the ILP and MLP of an application, the extra out-of-order logic comes at a price. Not only is the design more complex, requiring additional effort throughout the design cycle, out-of-order processors are also power-hungry. This further motivates the popularity of superscalar in-order processors, which are more power-efficient, in embedded systems.

To provide some insight into the differences between out-of-order and in-order processor architectures, we will now describe the flow of an in-

[4]The Bonnell microarchitecture is present in the popular Atom Z6 series.

struction from fetch to writeback on an out-of-order processor.

Instruction Fetch Similar as for in-order processors, instruction fetch loads instructions out of the memory by accessing the L1 instruction cache. It does this sequentially unless the branch predictor or execution unit redirects the fetch unit to fetch from a different address. The branch predictor is typically more complicated for out-of-order processors because more speculative instructions could be in-flight in the processor. Further, the cost of a mispredicted branch is relatively higher for out-of-order processors than for in-order processors, even when the front-end has the same number of pipeline stages. This is caused by the branch resolution time [23]: the time it takes to calculate the outcome of the branch.

Instruction Decode As with in-order processors the decode unit/stage identifies the type of work that needs to be done and sets up the signals to process the instruction further. Unlike in-order processors, the detection of inter-instruction dependencies does not happen in this stage but instead happens in the rename unit and the reservation station.

Instruction Rename The rename unit is a hardware structure that helps to allow out-of-order execution of instructions. It determines dependencies between instructions and maps the typically small architectural register file to a larger physical register file. This allows instructions that write the same output register (i.e., write-after-write (WAW) dependencies) to be scheduled out-of-order without harming program behavior. It also removes write-after-read (WAR) dependencies. Lastly it performs the necessary bookkeeping when an instruction leaves the ROB.

Dispatch The renamed instruction now gets dispatched in the ROB and in the reservation station. Depending on the microarchitecture, there could be multiple reservation stations (e.g., one assigned to each functional unit) and dispatch will need to ensure that the instructions are sent to the correct reservation station. Dispatch is, with the exception of the commit stage, the last stage where instructions flow in program order. The ROB keeps track of this order so that, even if instructions get executed out-of-order beyond dispatch, the microarchitectural state gets updated in-order, i.e., the instructions leave the microarchitecture in program order.

Issue In the issue stage, instructions get selected from the reservation station(s) and are sent to the various functional units. This is done by an instruction scheduler. For an instruction to get selected by the scheduler, a

necessary criteria is to have its source registers ready (i.e., all register dependences are resolved) and a functional unit to be available. In addition, more selection criteria, such as the age of the instruction, can be implemented in the scheduler.

Execute Similar to in-order processors, this stage starts calculation of the result of an instruction, using the appropriate functional unit. For instructions requiring access to the memory subsystem, an address generator unit (AGU) is often added to calculate memory addresses and detect memory dependencies. As an example, store-to-load forwarding logic could be added to forward results from a store instruction to a dependent load instruction.

Memory Memory instructions that have generated an address can access the memory in this stage. Because of the out-of-order nature of executing instructions, additional logic must be available to ensure the correct ordering of memory accesses to the same address. For example, when an instruction reads data from a memory address that is supposed to be written by an earlier instruction (i.e., a read-after-write dependency or RAW dependency) the microprocessor must make sure that the write happens before data gets read.

Writeback As mentioned before, instructions update the microarchitecture in program order, despite the fact that they might be executed out-of-order.

Commit Instructions leave the ROB in program order.

Examples of out-of-order microarchitectures are AMD's Jaguar, ARM's Cortex-A57, Qualcomm's Krait, Intel's Nehalem, etc.

2.2 Analytical Performance Modeling

Before designing the complex RTL code of a superscalar microprocessor, appropriate design decisions need to be made. This is a challenging process and is typically approached by running many cycle-level simulations. As mentioned in Chapter 1, cycle-level simulators execute applications at a speed of up 4 orders of magnitude slower than execution on real hardware. This slows down design space exploration as computer architects need to wait on simulation results. To accommodate for the slow speed, a better approach exists in having a good first-order analytical model which

trades off accuracy for speed. There are basically three approaches to analytical performance modeling: mechanistic modeling, empirical modeling and hybrid mechanistic/empirical modeling. In the next subsection we will briefly describe the different types of analytical modeling and give an overview of some of the existing related work.

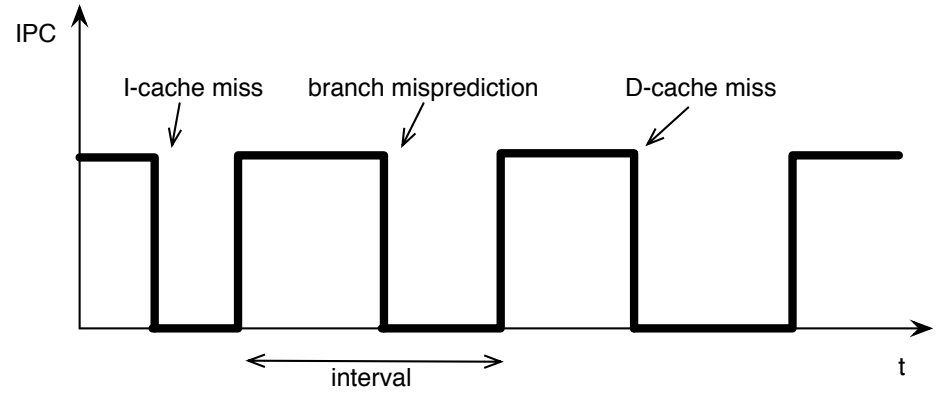
2.2.1 Mechanistic Modeling

Mechanistic modeling is derived from the actual mechanisms in the processor. A mechanistic model has the advantage of directly displaying the performance effects of individual mechanisms, expressed in terms of program characteristics such as inter-instruction dependence profiles and fine-grained instruction mix; machine parameters such as processor width, number of functional units and pipeline depth; and program-machine interaction characteristics such as cache miss rates and branch misprediction rates.

Prior work focused on mechanistic modeling of out-of-order processor performance for the most part. Michaud et al. [48] build a mechanistic model of the instruction window and issue mechanism. Karkhanis and Smith [41] extend this simple mechanistic model to build a complete performance model that assumes sustained steady-state issue performance punctuated by miss events. Chen and Aamodt [8] improve upon this model through more accurate modeling of pending data cache hits, overlaps between computation and memory accesses, and the impact of a limited number of MSHRs. Taha and Wills [59] propose a mechanistic model that breaks up the execution into so-called macro blocks, separated by miss events. Eyerman et al. [24] propose the interval model for superscalar out-of-order processors.

Whereas all of this prior work focused on out-of-order processors, in Chapter 3 we propose a mechanistic model for superscalar in-order processors. The fundamental assumptions made for modeling out-of-order processors do not hold true when modeling in-order processors. In [24, 41], a mechanistic model for out-of-order processors is built using interval analysis. Interval analysis is based on the observation that in the absence of miss events such as cache misses and branch mispredictions, a well-balanced superscalar out-of-order processor can smoothly stream instructions through its pipelines, buffers and functional units. Under ideal conditions the processor sustains a level of performance (instructions per cycle) roughly equal to the width of the processor. However, the smooth streaming of instructions is intermittently disrupted by miss events. The effects of these miss events divide execution time into intervals, and these intervals serve as the entity for analysis and modeling, see Figure 2.3(a) where performance is plotted as Instructions Per Cycle (IPC) over time.

(a) Interval modeling of superscalar out-of-order processor performance



(b) Interval modeling of superscalar in-order processor performance

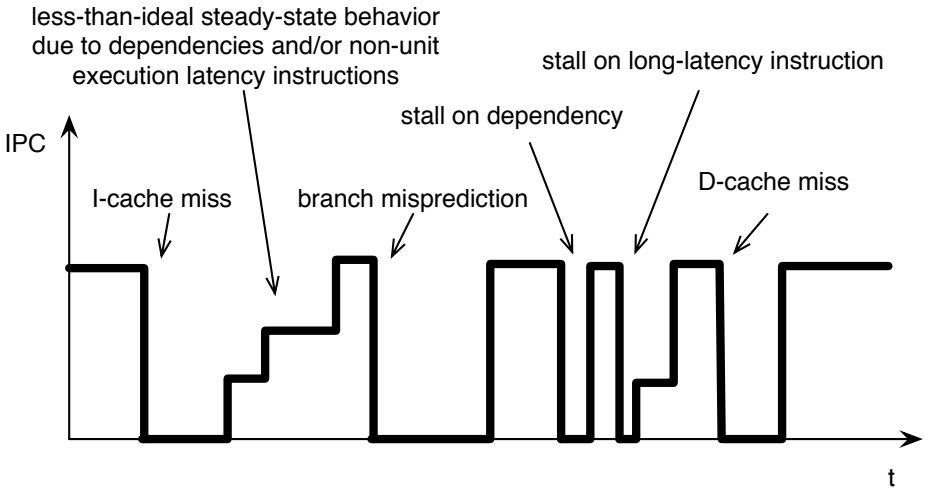


Figure 2.3: Interval analysis analyzes processor performance on an interval basis determined by disruptive miss events: (a) out-of-order processors and (b) in-order processors.

The fundamental assumption made for modeling superscalar out-of-order processors, namely that the processor can smoothly stream instructions through its pipelines at roughly the designed width, does not hold true for in-order processors. Moreover, superscalar out-of-order processors can be modeled by taking a coarser level of miss events into account, such as long-latency cache misses (typically last-level cache misses only due to data references), instruction cache misses, TLB misses, and branch mispredictions. In-order processors on the other hand, incur a wider range of miss events and other performance hazards. Beyond the ones mentioned

above, in-order processor performance also suffers from pipeline stalls due to inter-instruction dependences, functional unit contention, non-unit instruction latencies and cache misses in first-level cache(s). (An out-of-order processor is designed such that these latencies and inter-instruction dependences are mostly hidden.) As a result, inter-instruction dependences, functional unit contention and non-unit instruction execution latencies may introduce additional intervals and in addition may lead to a pipeline throughput that is less than the designed width in the absence of miss events, see Figure 2.3(b). These fundamental differences make it impossible to exactly mimic the behavior of an in-order processor by constraining out-of-order resources (e.g., the ROB size). The mechanistic model proposed in Chapter 3 models the additional phenomena using program statistics, such as a fine-grained instruction mix and inter-instruction dependence profiles, that are independent of the underlying machine.

2.2.2 Empirical Modeling

In contrast to mechanistic modeling, empirical modeling requires little or no prior knowledge about the system being modeled: the basic idea is to learn or infer a performance model using machine learning and/or statistical methods from a large number of detailed cycle-accurate simulations. Empirical modeling seems to be the most widely used analytical modeling technique today, and was employed for modeling out-of-order processors only, to the best of our knowledge. Some prior proposals consider linear regression models for analysis purposes [39]; non-linear regression for performance prediction [38]; spline-based regression for power and performance prediction [45]; neural networks [16, 36]; or model trees [50].

2.2.3 Hybrid mechanistic-empirical modeling

Hybrid mechanistic-empirical modeling targets the middle ground between mechanistic and empirical modeling: starting from a generic performance formula derived from understanding the underlying mechanisms, unknown parameters are derived by fitting the performance model against detailed simulations. For example, Hartstein and Puzack [31] propose a hybrid mechanistic-empirical model for studying optimum pipeline depth; the model is tied to modeling pipeline depth only and is not generally applicable. Eyerman et al. [25] proposed a more complete mechanistic-empirical model which enables constructing CPI stacks on real out-of-order processors.

2.3 Workload selection techniques

It is well-known that having representative benchmarks is key to design space explorations, as using non-representative benchmarks may lead, or are likely to lead, to suboptimal design points. As a result, substantial prior work has been done towards identifying representative benchmarks. In addition, to reduce simulation time, many researchers have worked on techniques toward finding representative samples within benchmark applications. To the best of our knowledge, however, no comprehensive study was previously published regarding the impact of benchmark inputs on design space exploration.

We now discuss some of the prior work in benchmark and sample selection, followed by a description of prior work in generating and evaluating benchmark inputs.

2.3.1 Benchmark Selection

Common practice when composing a benchmark suite is to identify key applications in a given application domain of interest, from which benchmarks and inputs are then selected. An important requirement for benchmark suites to be shared across parties (academia and/or industry) is that the benchmarks are open source, although Non-Disclosure Agreements (NDAs) may enable processor manufacturers to use proprietary customer workloads to evaluate future designs. Example open-source benchmark suites are SPEC CPU [32] for general-purpose computing, DaCapo [4] for Java workloads, PARSEC [2] for multi-core, Rodinia [7] for heterogeneous CPU/GPU systems, MiBench [28] for embedded workloads, CloudSuite [26] for cloud workloads, etc. The benchmark selection typically involves making sure the benchmark suite, as a whole, covers the most important application behaviors in the target domain, while being portable enough to use across different platforms.

A number of papers have been published to evaluate the representativeness of a benchmark suite and/or to select a representative subset from a larger pool of benchmarks. Eeckhout et al. [19] propose data analysis (Principal Components Analysis) and machine learning (cluster analysis) techniques to identify a diverse and representative benchmark subset. Follow-on work used microarchitecture-independent benchmark characterization as input to this methodology [40], or a number of real hardware measurements [52]. Yi et al. [68] use a Plackett-Burman design of experiment in which they measure performance on a number of different processor architectures to understand how benchmarks interact differently with microarchitecture parameters. They then pick diverse benchmarks using cluster analysis to cover the workload space as much as possible.

SubsetTrio [37] translates the benchmark selection problem into a geometrical problem and uses the notion of a convex hull to identify most diverse benchmarks — the benchmarks at the outer range of the benchmark space — in contrast to cluster analysis which groups benchmarks into clusters of similar benchmarks, and then picks a representative benchmark per cluster. Yi et al. [69] quantify the pitfall of using non-representative, old benchmarks to design future processors. They discuss a case study in which they identify the optimum processor for SPEC CPU95, and then evaluate this processor using CPU2000. They report an EDP deficiency of 18.5%. This, once more, underlies the importance of using representative benchmarks during design space exploration.

While composing representative benchmark suites is challenging for single-core experiments, it is a daunting task for multi-core and multi-threaded processors. Such processors can run multiple independent thread contexts, which leads to an explosion in the number of possible multi-program workloads [63, 64, 66] and combinations of benchmark starting points [62, 54].

2.3.2 Sample Selection

An additional concern to finding representative benchmarks, is to find regions within a benchmark's execution that are representative for the entire benchmark execution. SimPoint [57] collects Basic Block Vectors (BBVs) on a per-region basis and uses cluster analysis to find representative regions within a benchmark execution. Eeckhout et al. [21] use microarchitecture-independent characterization to identify representative regions across benchmarks. Van Biesbrouck et al. [61] introduce the Co-Phase Matrix to find the most representative regions in multi-program workloads.

2.3.3 Input Selection

As mentioned before, the amount of work done in characterizing the impact of input data sets on design space exploration is limited. KleinOsowski and Lilja [42] proposed reduced input sets for SPEC CPU2000 to reduce simulation time. Later work by Eeckhout et al. [20] found these reduced inputs to be representative for some benchmarks, but not for others.

Chen et al. [10] propose KDataSets, a set of 1,000 data sets for a broad set of the MiBench benchmarks. They use KDataSets to understand data set sensitivity for iterative optimization which aims at finding the best set of compiler optimizations for a given application. They find that some input data sets have a deviation of 10 to 25% from the average speedup across data sets. Further, they reveal how different input data sets react differ-

ently on the same set of compiler optimization flags and come up with a strategy to find a single set of compiler flags to generate a program binary that achieves nearly-optimal speedups for all input data sets.

We use KDataSets in Chapter 4 to study how sensitive processor customization is with respect to application inputs. As we show in Chapter 4, design space exploration using a badly chosen application input can result in poor design decisions. We therefore introduce three novel techniques to help computer architects select representative application inputs when performing microarchitectural desgin space exploration.

2.3.4 Summary

In this chapter we discussed a number of background topics and related work. We briefly discussed some of the important microarchitectural structures in modern superscalar in-order and out-of order processors. The main difference between out-of-order and in-order processors is that out-of-order processors can execute instructions in an order which is different than the sequence specified by the program. This is made possible by providing the out-of-order processor with additional logic. While this additional logic improves performance by exploiting instruction-level and memory-level parallelism, it comes at the cost of a more complex and power-hungry design.

We further discussed some related work on mechanistic modeling of superscalar out-of-order processors and revealed some of the difficulties with modeling in-order processors. Because out-of-order processors are built to hide latencies of inter-instruction dependences, long-latency instructions and functional unit contention, their performance is only weakly affected by these events. In-order processors on the other hand, suffer from the frequent stalls caused by these events, and hence additional modeling is required. In Chapter 3 we introduce our mechanistic model, which uses fine-grained information such as inter-instruction dependence distances and instruction mixes on very small instruction groups to model the additional penalty caused by these events. Next to related work on mechanistic modeling we also discussed the two other main approaches in analytical performance modeling, namely empirical and hybrid mechanistic-empirical modeling.

We end this chapter with a discussion on related work on workload selection techniques. Much of the prior work focuses on minimizing the number of benchmarks, while still covering the workload space as much as possible. In order to limit the long execution time on slow cycle-level simulators, prior work has focused on finding representative regions within benchmarks. We further discussed the limited amount of work done in characterizing the impact of input data sets on design space exploration.

Until now, the assumption was to use one or a couple of benchmark inputs to drive design space exploration. In Chapter 4 we quantify the potential pitfall of selecting incorrect benchmark inputs and introduce input selection techniques to find representative benchmark inputs.

Chapter 3

Mechanistic Analytical Performance Modeling of Superscalar In-order Processors

Everything should be made as simple as possible,
but not simpler.
Albert Einstein

In this chapter we present a fast off-line analytical performance evaluation model for superscalar processors. The intent of the model is to help computer architects get an estimate of an application's performance on a wide range of microarchitectural designs. Off-line evaluation is in the order of milliseconds, making it a very appealing complementary exploration tool to slow cycle-level architectural simulations, which can take hours to days.

The construction of the model is based on the internal mechanics of the micro-processor. Constructing a model for superscalar in-order processors is not trivial. The performance of a processor is subject to many events that occur throughout execution: several types of miss events such as cache misses and branch mispredictions can stall the processor for multiple cycles. Furthermore, in-order processors execute instructions in program order. This gives rise to additional events that stall the processor, namely inter-instruction dependences (i.e., an instruction consuming the outcome of a previous instruction) and functional unit contention (i.e., multiple instructions requiring the same functional unit). These additional events can be hidden on out-of-order processors, which relaxes the modeling and allows to evaluate the processor with interval modeling [24]. For in-order

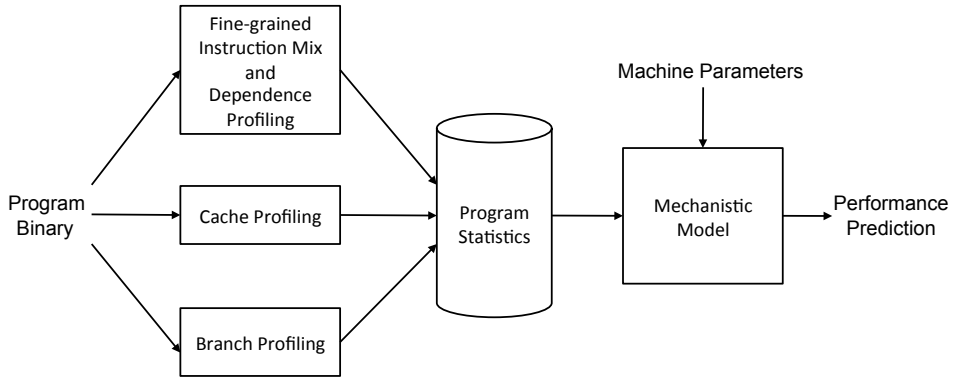


Figure 3.1: Overview of the mechanistic modeling framework.

processors, inter-instruction dependences and functional unit contention have a large impact on performance and require fine-grained profile information and modeling in order to calculate an accurate performance estimate.

This chapter is organized as follows. Sections 3.1 to 3.4 discuss the construction of the model. In Section 3.5 we detail the experimental setup in which we validate and use the model. The model is validated against both detailed simulation and hardware in Section 3.6. We illustrate the usage of the model through use cases related to design space exploration in Section 3.7. Further we show how we can get insights in the application-microarchitecture interaction in Section 3.8. Finally we conclude this chapter in Section 3.9.

3.1 Modeling context

Before describing the proposed model in great detail, we first set the context within which we build the model. We present a general overview of the modeling framework, as well as a description of the assumed superscalar in-order processor architecture.

3.1.1 General overview

The framework of the mechanistic model is illustrated in Figure 3.1: it requires a profiling run to capture a number of statistics that are specific to the program only and that are independent of the machine. These statistics relate to the program's instruction mix and inter-instruction dependences, and need to be collected only once for each program binary.

The profiling run also needs to collect a number of mixed program-

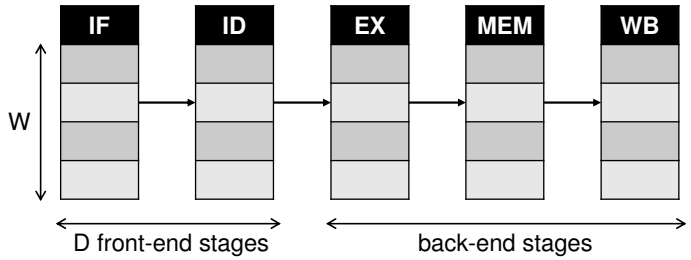


Figure 3.2: Schematic view of the assumed superscalar in-order processor. Here, W=4 and D=2.

machine statistics, i.e., statistics that are a function of both the program binary as well as the machine configuration. Example statistics are cache and TLB miss rates, and branch misprediction rates. Although, in theory, collecting these statistics requires separate runs for each cache, TLB and branch predictor configuration of interest, in practice though, most of these statistics can be collected in a single run. In particular, single-pass cache simulation [58] allows for computing cache miss rates for a range of cache sizes and configurations in a single run. We also collect branch misprediction rates for multiple branch predictors in a single run. Once these statistics are collected, we can predict cache miss rates and branch misprediction rates for any combination of cache hierarchy with any branch predictor and any processor core configuration.

These statistics, along with a number of machine parameters, serve as input to the analytical model, which then estimates superscalar in-order processor performance. The machine parameters include pipeline depth, pipeline width, number of functional units and their types, functional unit latency (multiply, divide, etc.), cache access latencies, and memory access latencies; further, the cache/TLB and branch predictor size and configuration of interest needs to be selected.

Because the analytical model basically involves computing a limited number of formulas, a performance prediction is obtained almost instantaneously. In other words, once the initial profiling is done, the analytical model allows for predicting performance for a very large design space in the order of seconds or minutes at most.

3.1.2 Microarchitecture description

As shown in Figure 3.2, we assume a superscalar in-order processor with five pipeline stages: fetch (IF), decode (ID), execute (EX), memory (MEM) and write-back (WB). Fetch and decode are referred to as the front-end stages of the pipeline, whereas execute, memory and write-back are back-end stages. Note we consider a five stage pipeline without loss of gen-

erality; we can still model deeper pipelines by considering longer front-end pipelines, and non-unit latency instruction execution units, as will become clear later. Each stage has W slots (numbered from 0 to $W-1$) to hold a total of W instructions, with W being the width of the processor. We assume forwarding logic such that dependent instructions can execute back-to-back in subsequent cycles. Further, we assume stall-on-use, i.e., the processor stalls on an instruction that consumes a value that has not been produced yet. These instructions block in the ID-stage. Load instructions perform address calculation in the EX-stage and perform the cache access in the MEM-stage. Finally, we assume in-order commit to enable precise interrupts. This implies that instructions that take more than one cycle to execute (e.g., a multiply instruction or a cache miss) block all subsequent instructions from going to the WB-stage. Since each stage can only hold W instructions, this further implies that when a long-latency instruction blocks instructions from passing from the MEM-stage to the WB-stage, the EX-stage will eventually be filled with instructions and hence no instructions can leave the ID-stage.

3.2 Overall formula

The overall formula for estimating the total number of execution cycles T of an application on a superscalar in-order processor is as follows:

$$T = \frac{N}{W} + P_{misses} + P_{deps} + P_{FU}. \tag{3.1}$$

In this equation, N equals the number of dynamically executed instructions; W stands for the width of the processor; P_{misses} is the total penalty due to miss events; P_{deps} is the total penalty due to inter-instruction dependences; and P_{FU} stands for the penalty due to functional unit limitations (i.e., structural hazards).

The intuition behind the mechanistic model is that the minimum execution time for an application equals the number of dynamically executed instructions divided by processor width, i.e., it takes at least N/W cycles to execute N instructions on a W-wide processor in the absence of miss events and stalls. Miss events, inter-instruction dependences and functional unit contention prevent the processor from executing instructions at a rate of W instructions per cycle, which is accounted for by the model by adding penalty cycles.

The next sections discuss each of the terms of the formula. We start with miss event penalties and then discuss instruction dependences and functional unit contention.

3.3 Miss events

We determine the penalty due to miss events using the following formula:

$$P_{misses} = \sum_{i \in \{missEvents\}} misses_i \times penalty_i. \tag{3.2}$$

This formula computes the sum over the miss events, weighted with their respective penalties. We make a distinction between cache (and TLB) misses and branch mispredictions when it comes to computing the penalties.

3.3.1 Penalty due to cache and TLB misses

When an instruction cache miss occurs, the instructions in the front-end pipeline can still enter the execution stage, but when the instruction cache miss is resolved, it takes some time for the new instructions to re-fill the front-end pipeline. It is easy to understand that the front-end pipeline drain time and re-fill time offset each other, i.e., the penalty for an instruction cache miss is independent of the front-end pipeline depth. In case of a data cache miss, the memory stage blocks, and no instructions can leave or enter the execution stage until the data cache miss is resolved.

From the above discussion, it follows that the penalty for a cache miss equals its miss latency [1]. However, when a cache miss occurs, it might be the case that some instructions can complete execution in parallel with the miss penalty. In case of an instruction cache miss on a 4-wide processor, it may happen that one, two or three instructions were already fetched before the instruction cache miss occurred. Similarly, for a data cache miss, depending at which slot the load instruction enters the MEM-stage in Figure 3.2, it may happen that one (the load instruction enters the MEM-stage at the second slot), two (the load instruction enters at the third slot) or three older instructions (the load instruction enters at the last slot) proceed to the WB-stage. These instructions can complete underneath the cache miss, and are therefore hidden. Assuming that cache misses are uniformly distributed across a W-wide instruction group, the average number of instructions hidden underneath a cache miss equals $\frac{W-1}{2}$. This means that the miss penalty can be reduced by $\frac{W-1}{2W}$ cycles (which is less than one cycle). The total penalty for a cache or TLB miss thus equals

$$penalty_{cacheMiss} = MissLatency - \frac{W-1}{2W}. \tag{3.3}$$

[1] For caches the miss latency is the access time to the next level of cache or main memory. For TLBs the miss latency is the time it takes for the page table walk to refill the TLB entry.

Memory-level parallelism Memory-level parallelism (MLP) is defined as the number of simultaneously outstanding misses if at least one is outstanding [11]. This implies that we only have to account for the first, non-overlapped memory access latency as independent memory accesses later in the instruction stream are hidden underneath the first access. Out-of-order processors make use of this property by implementing a reorder buffer and Miss Status Handling Registers (MSHRs) to exploit memory-level parallelism over a large window of instructions. For in-order processors, on the other hand, this window is limited to both the width of the processor, and the distance to the first instruction in the dynamic instruction stream that depends on the load miss (stall-on-use). The shorter window size for in-order processors in which MLP can be exploited results in a lower amount of MLP than for out-of-order processors. However, for a number of benchmarks we observe MLP values significantly greater than one. When taking MLP into account the penalty associated with the cache miss term in Formula 3.2 gets scaled as in Formula 3.4 below:

$$P_{cacheMisses} = \frac{cacheMisses_i}{MLP} \times penalty_{cacheMiss}. \tag{3.4}$$

As described by Van Craeynest et al. [65], we can calculate the MLP as the number of memory accesses between a load instruction and its first consumer, since this consumer blocks the ID-stage (stall-on-use). However, since the processor can only hold W instructions per pipeline stage, the load instruction will block any instruction at a distance larger than $(W-1)$ instructions from proceeding to the next stage. This implies that we never need to account for memory accesses outside of a window larger than W instructions, even if the first consumer of the load is further than $(W-1)$ instructions apart. We have implemented a simple profiler that determines the average dependence distance between a load and its first consumer with the inter-instruction dependence profile. We combine this with the fine-grained instruction mix profile to count the number of independent load instructions within this distance.

3.3.2 Penalty due to branch mispredictions

Branch mispredictions are slightly different than cache misses. Upon a branch misprediction, all the instructions fetched after the mispredicted branch need to be flushed. In particular, when a branch misprediction is detected in the execution stage, all the instructions in the front-end pipeline as well as the instructions fetched after the branch in the execute stage need to be flushed. Hence, the penalty of a branch misprediction equals:

$$penalty_{branchMiss} = D + \frac{W-1}{2W}, \tag{3.5}$$

with D the depth of the front-end pipeline. The first term is the number of cycles lost due to flushing the front-end pipeline: there are as many cycles lost as there are front-end pipeline stages, namely D. The second term is the penalty of flushing instructions in the execute stage. This number ranges between 0 and $W-1$ and depends on the number of instructions in the execute stage that are younger than the branch. We approximate this as the average number of instructions in the execute stage, younger than the branch instruction, under a uniform distribution of instructions in the pipeline stage.

Correctly predicted branches may also introduce a performance penalty. In our setup, a branch is predicted one cycle after it was fetched, and if it is predicted taken, the instruction(s) in the fetch stage and the instructions in the decode stage that are younger than the branch (which were fetched assuming a non-taken branch) need to be flushed, incurring a pipeline bubble. This incurs $1 + \frac{W-1}{2W}$ penalty cycles per branch that is predicted taken, even if it is correctly predicted. We will refer to this penalty as the taken-branch hit penalty.

3.4 Inter-instruction dependences and functional unit contention

To determine the penalty caused by inter-instruction dependences and functional unit contention, we need to keep track of (1) the distance between dependent instructions (the smaller the distance, the more likely the processor will stall to resolve the dependence); and (2) the order in which different instructions execute (subsequent instructions of the same type will put more pressure on the specific functional unit). In many cases instructions will suffer from both dependences on previous instructions and from contention of functional units. We therefore summarize this information collectively in the pattern history distribution matrix (H-matrix), which we will use to calculate the penalty caused by inter-instruction dependences and functional unit contention. Before explaining the formula, we will first illustrate how the H-matrix is constructed. Each instruction in the dynamic instruction stream can be represented by recording the history of the types of the $W-1$ previous instructions, which we call a *pattern*, together with the distance to the closest instruction it depends on. We can use this pattern i and dependence distance j as row and column indexes, respectively, to increment a counter in the H-matrix for each instruction. As a result, the elements of the H-matrix represent counters that indicate the occurrences of patterns i with a dependence distance of j [2]. This can

[2]Note that the H-matrix can be reused for any microarchitectural configuration with a superscalar width $\leq W$. This implies that collecting the H-matrix is a one-time cost only.

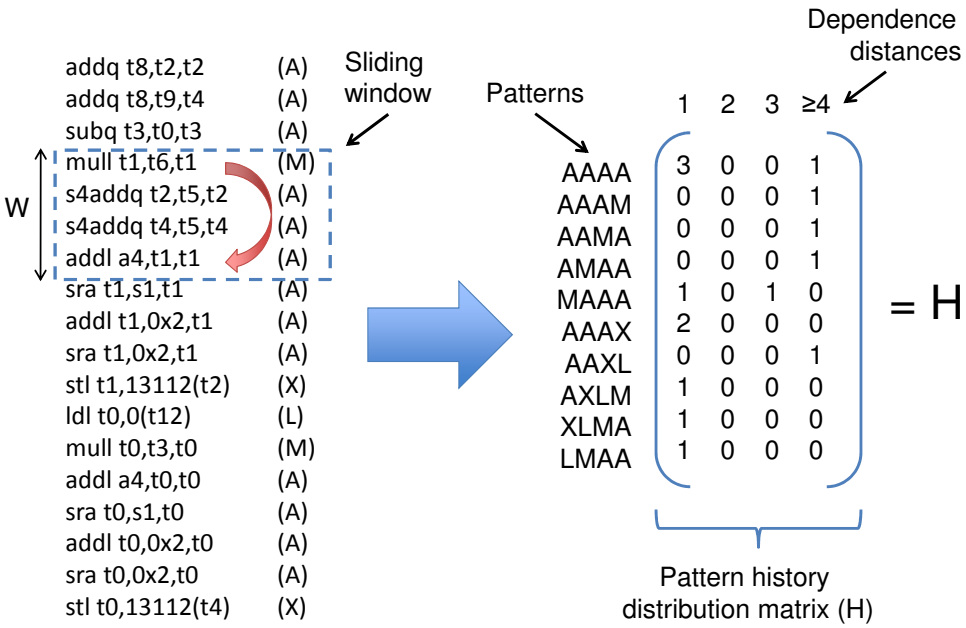


Figure 3.3: Construction of the pattern distribution matrix for part of the dct_chroma routine of the h264 benchmark.

best be illustrated with an example. Figure 3.3 shows a small portion of the dynamic instruction flow of the dct_chroma routine of h264 on the left, together with its associated H-matrix on the right.

For the ease of visualization, each assembly instruction is represented by a symbol which indicates the type of instruction: A stands for an ALU-instruction, M for a multiply/divide, L for a load, and X for all other instructions that are not needed for the penalty calculation (store instructions, branches, etc.)[3]. To construct the H-matrix we make use of a sliding window that slides through the whole instruction stream, instruction by instruction. The size of the window equals the maximum processor width W of interest, which we set to 4 here and in all subsequent examples (unless stated otherwise). For each last instruction in the window we record the distance to the closest instruction it depends on together with the history of preceding instructions in the order executed from old (left) to most recent (right). Consider the position of the sliding window in Figure 3.3. The last instruction in the window is an ALU-instruction and the history pattern of instructions is denoted by "MAAA". The last instruction has a dependence on the multiply instruction at distance 3 (as indicated by the red arrow). Therefore we increment the H-matrix counter at the row with

[3]X stands for don't care instructions, as these instructions don't contribute to the penalty calculation for dependences and functional unit contention. Note that in the remainder of the paper we will mark other instructions irrelevant for the calculation with X.

pattern "MAAA" and the column that represents dependence distance 3. We can now shift the window one instruction down to increment a counter for pattern "AAAA" at distance 1. We continue this process for all instructions.

By associating a penalty term to each row and column in the H-matrix we can determine a cost matrix C. The C-matrix represents the cost (in number of cycles) a specific history/dependence-pattern incurs on the performance of the processor. By multiplying the matrices C and H termwise and by accumulating them (i.e., taking the Frobenius product), we can calculate the total penalty term for functional unit contention and inter-instruction dependences, as shown in Formula 3.6

$$P_{deps} + P_{FU} = \sum_{i \in patterns, d=1..2\times W} C_{i,d} \times H_{i,d} = C : H \tag{3.6}$$

To determine the individual terms in the C-matrix (i.e., the cost associated with a specific history-pattern and dependence distance), we first need to determine the penalty for the specified history-pattern in case there were no dependences and vice versa the penalty for the specified dependence distance, assuming a sufficient number of functional units. In case an instruction waits both for a dependence to resolve and for a functional unit to become available, the largest of those two penalties will be accounted for, as shown in Formula 3.7.

$$C_{i,d} = \max\left(c_{dep}(i,d), c_{fu}(i)\right) \tag{3.7}$$

$c_{dep}(i,d)$ represents the penalty of pattern i when the last instruction in the pattern has a dependence at distance d. $c_{fu}(i)$ is the penalty caused by functional unit contention for pattern i, i.e., when multiple instructions from the same instruction type reside in pattern i, we need to account penalty waiting for an appropriate unit to start processing the last instruction of pattern i. The terms $c_{dep}(i,d)$ and $c_{fu}(i)$ are subject of the next subsections.

Although Formula 3.6 shows how the sum of dependence penalties and functional unit contention penalties can be calculated together, we will show in Section 3.8.1 how we can determine these penalties separately.

3.4.1 Inter-Instruction Dependences

Dependences on unit-latency instructions

In this section we derive the penalty of an instruction that depends on the outcome of a close-by (within W instructions) unit-latency instruction. In our setup, integer ALU instructions are the only unit-latency instructions,

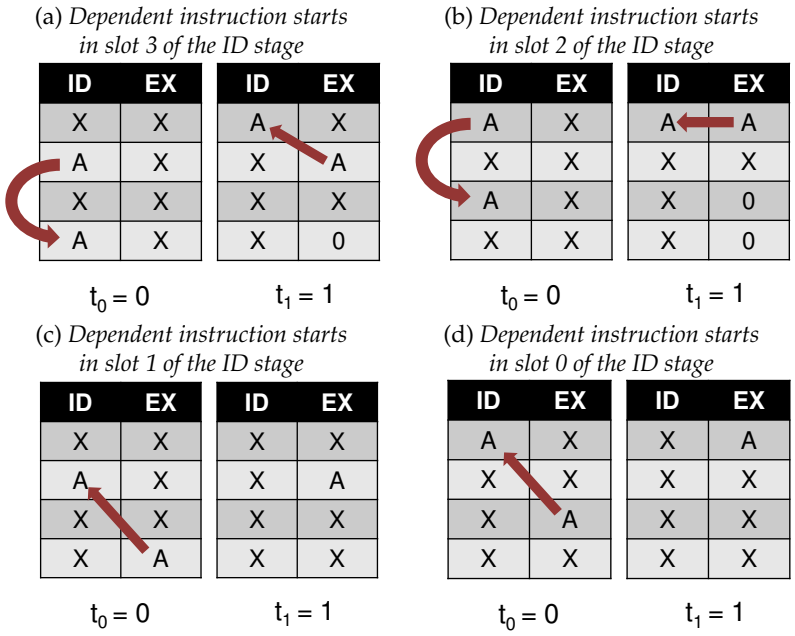


Figure 3.4: Four possible instruction flows for an instruction dependent on an ALU instruction at distance $d = 2$.

so for the remainder of this section, we will refer to them as ALU instructions. Dependences on an ALU instruction resolve the cycle after the ALU instruction gets executed.

To illustrate the penalty calculation, consider the example pattern "XAXA" where the first (oldest) ALU instruction produces a data value that is consumed by the next ALU instruction, i.e., the dependence distance d equals 2. There are 4 possible positions at which the instructions can enter the ID-stage, see the ID-stage at t_0 in the four subfigures of Figure 3.4. (The oldest instruction in an instruction group is shown at the top of each pipeline stage, and '0' denotes a bubble or an empty slot due to a stall.) In case (a), the dependent instruction enters the ID-stage at slot 3. When the ALU instruction starts execution at t_1, the dependent instruction gets blocked because the result of the ALU instruction is not available yet. Since the dependent instruction is the last instruction in the ID-stage at t_0, one slot will be unused in the EX-stage at t_1 (marked with 0), hence we lose $\frac{1}{4}$ of a cycle. In case (b), the dependent instruction enters the ID-stage in slot 2 at t_0. As in case (a), it again gets blocked from going to the EX-stage at t_1. Next to blocking itself it also blocks the younger instruction that was at slot 3 in the ID-stage of t_0. This means that we now lose $\frac{2}{4}$ of a cycle. In cases (c) and (d), the producing ALU instruction already started execution at t_0. This means that the dependence is resolved by the time the dependent instruction starts execution and hence no cycles are lost.

Assuming the four situations have equal probability to occur (uniform distribution of the instructions in the pipeline), we derive the penalty for an instruction dependent on an ALU instruction at distance 2 as follows:

$$\begin{aligned} c_{dep}(\mathrm{XAXA}, 2) &= (\mathrm{Prob}[\mathrm{pos}=0] + \mathrm{Prob}[\mathrm{pos}=1]) \times 0 \\ &\quad + \mathrm{Prob}[\mathrm{pos}=2] \times \frac{2}{4} + \mathrm{Prob}[\mathrm{pos}=3] \times \frac{1}{4} \\ &= \frac{1}{4} \times 0 + \frac{1}{4} \times 0 + \frac{1}{4} \times \frac{2}{4} + \frac{1}{4} \times \frac{1}{4} = \frac{3}{16}. \end{aligned} \tag{3.8}$$

In general, we can calculate the penalty for an instruction dependent on an ALU instruction at distance d (with $d < W$) using the following formula:

$$c_{dep}(i, d < W) = \sum_{j=0}^{W-1} Prob\left[pos = j | pat = i\right] \times \begin{cases} \frac{W-j}{W} & \text{if} \quad d \leq j \\ 0 & \text{else} \end{cases}$$

$$= \sum_{j=d}^{W-1} \frac{1}{W} \times \frac{W-j}{W} \tag{3.9}$$

$$= \frac{(W-d)\,(W-d+1)}{2W^2}. \tag{3.10}$$

In this equation, i represents the pattern, W the processor width, d the dependence distance to the closest ALU instruction, and j the slot of the newest instruction in the decode stage. $Prob\left[pos = j | pat = i\right]$ represents the probability that the newest instruction in pattern i is at slot j in the decode stage. Note that i can be any pattern, with the constraint that the $(d+1)$'th symbol is an 'A'-symbol (i.e., the producer is an ALU instruction). The inequality $j \geq d$ indicates that we only need to account penalty when the producing ALU instruction is in the ID-stage at the same cycle. In Formula 3.9, we make use of the assumption that instructions are uniformly distributed in the pipeline stage.

Special case: non-uniform distribution of instructions in the pipeline We find the assumption of instructions to be uniformly distributed in the pipeline stage to be accurate for the set of benchmarks used in our setup. However, one could design a corner case application that is dominated by instructions with dependence distances of 1 that causes all instructions to be serialized. Hence, most of the instructions of this application will enter the first slot of the ID-stage. To model these corner cases we could estimate the probabilities with a heuristic based on the overall average dependence distance: if the average dependence distance is close to 1 more weight needs to be given to $Prob\left[pos = 0\right]$. However, we found this case to be very rare in our setup, and modeling it increases the complexity of the model without noticeably improving its accuracy.

Dependences on load instructions

Unlike ALU instructions, load instructions do not see their result in the EX-stage but in the MEM-stage. This has two consequences for calculating the penalty caused by instructions dependent on a load instruction. First, this means that if a load instruction and its consumer reside in the ID-stage in the same cycle, an additional penalty cycle will need to be accounted for on top of the one calculated with Formula 3.10. Second, even when the load instruction and its dependent instruction reside in consecutive stages (load in the EX-stage, dependent instruction in the ID-stage), a penalty needs to be accounted for.

When the distance between the load and the dependent instruction is $0 < d < W$, the load instruction can either be in the same stage or in a consecutive stage as the dependent instruction. When $W \leq d < 2W$, the load instruction and the dependent instruction can never reside in the same stage, and hence we will only need to account penalty when they reside in consecutive stages.

The reasoning for calculating the penalty when $W \leq d < 2W$ is fairly similar as for dependences on ALU instructions where $d < W$. We can find the penalty for dependences on load instructions for a dependence distance $W \leq d < 2W$, by substituting d by $d - W$ in Formula 3.9:

$$c_{dep}(i, d \geq W) = \sum_{j=d-W}^{W-1} \frac{W-j}{W^2} \tag{3.11}$$

$$= \frac{(2W-d)(2W-d+1)}{2W^2} \tag{3.12}$$

For $0 < d < W$ the penalty depends on whether the load instruction and its consumer resided in the ID-stage during the same cycle or in subsequent stages. In the case they resided in different stages, the penalty depends on the position of the consumer in the ID-stage. Assuming again a uniform distribution of instructions in the pipeline stage the penalty can be calculated as follows:

$$c_{dep}(i, d < W, \text{subsequent stage}) = \sum_{j=0}^{W-1} \frac{W-j}{W^2} \tag{3.13}$$

If the load instruction and its consumer resided in the same stage we need to account for an additional penalty cycle:

$$c_{dep}(i, d < W, \text{same stage}) = \sum_{j=0}^{W-1} \frac{W-j}{W^2} + 1 \tag{3.14}$$

To know whether the load and its consumer resided in the ID-stage during the same cycle, we would need to know either in which slot the load

instruction or its consumer arrived in the ID-stage. We can estimate this based on the probability of how instructions are distributed in the pipeline stage. Assuming again a uniform distribution, we can combine Formulas 3.13 and 3.14 into Formula 3.16 as follows:

$$c_{dep}(i, d < W) = \sum_{j=0}^{W-1} \frac{W-j}{W^2} + \sum_{j=d}^{W-1} \frac{1}{W} \times 1 \quad (3.15)$$

$$= \frac{3W + 1 - 2d}{2W} \quad (3.16)$$

Dependences on long-latency instructions

Other long-latency instructions, such as integer multiply instructions wait in the MEM stage until their result is calculated. Therefore, instructions dependent on long-latency instructions are penalized the same way as instructions dependent on load instructions, by Formulas 3.16 and 3.12 for $0 < d < W$ and $W \leq d < 2W$, respectively. Note that the latency of the long-latency instruction itself will be accounted for as a functional unit penalty (see next section), so for the dependent instruction, we only have to account for the empty issue slots between the long-latency instruction and the dependent instruction.

Special case: long-latency dependences between instructions of the same type. As we will show in Sections 3.4.2 and 3.4.2, long-latency instructions can sometimes be executed in parallel, depending on the number of functional units available. As a result, no additional penalty is accounted to the instruction executing in parallel with another instruction. However, if there is a dependence between these instructions, the latency will not be hidden. We account for this by adding (part of) the latency of the specific instruction to the dependence penalty. Again we need to distinguish between $0 < d < W$ and $d \geq W$. For $d \geq W$, we can calculate the penalty as follows:

$$c_{dep}(i, d \geq W) = (\text{latency} - 2) + \frac{(2 \times W - d + 1) \times (2 \times W - d)}{2 \times W^2} \quad (3.17)$$

The intuition behind Formula 3.17 is that the long-latency instruction arrived in the ID-stage at least one cycle and at most two cycles earlier than the dependent instruction. Hence, at least (latency$-1 \times W$) and at most (latency$-2 \times W$) instructions cannot start execution because of the dependence, resulting in a penalty between (latency-1) and (latency-2) cycles.

The exact number depends on the position the long-latency instruction entered the ID-stage and the distance to the dependent instruction a number.

When $0 < d < W$ we again need to distinguish between whether both long latency instructions arrived in the ID-stage during the same cycle, or one cycle apart. When the consuming long latency arrives one cycle later we calculate the penalty as follows:

$$c_{dep}(i, d < W, \text{subsequent cycle}) = (\text{latency} - 2) + \sum_{j=0}^{W-1} \frac{W-j}{W^2} \tag{3.18}$$

When both of the instructions arrive during the same cycle, the penalty is calculated as follows:

$$c_{dep}(i, d < W, \text{same cycle}) = (\text{latency} - 1) + \sum_{j=0}^{W-1} \frac{W-j}{W^2} \tag{3.19}$$

Similar as with Formula 3.16, we can combine Formulas 3.18 and 3.19 into Formula 3.20 as follows, when assuming a uniform distribution of instructions in a pipeline stage:

$$c_{dep}(i, d < W) \quad = \quad (\text{latency} - 2) + \frac{3 \times W + 1 - 2 \times d}{2 \times W} \tag{3.20}$$

3.4.2 Functional Unit Contention

ALU contention

We first derive the penalty for integer ALU instructions due to functional unit contention. We model ALU contention penalties in a way that is analogous to dependence penalties on ALU instructions. For this we need to define an analogy to the dependence distance:

$d_U(i)$: distance to the instruction that causes functional unit contention when the processor has U units, for pattern i.

For example, for a superscalar processor of width $W = 4$ with 2 ALUs, the contention distance for pattern "XAAA", $d_U(\text{XAAA})$, equals to 2, because the last instruction in the group cannot execute immediately due to the fact that there are only 2 ALUs, and there are two ALU instructions preceding the last ALU instruction.

Replacing the dependence distance d with $d_U(i)$ in Formula 3.9, allows

us to determine the ALU contention penalty:

$$
\begin{aligned}
c_{fu}(i) = \mathrm{fr}(i) &= \sum_{j=0}^{W-1} Prob\,[pos = j | pat = i] \times \begin{cases} \frac{W-j}{W} & \text{if} \quad d_U(i) \leq j \\ 0 & \text{else} \end{cases} \\
&= \sum_{j=d_U(i)}^{W-1} Prob\,[pos = j | pat = i] \times \frac{W-j}{W} \qquad (3.21) \\
&\approx \frac{(W - d_U(i))(W - d_U(i) + 1)}{2W^2}. \qquad (3.22)
\end{aligned}
$$

In this equation i represents the pattern, W the processor width, $d_U(i)$ the distance to the closest instruction that causes contention when U units are present, as defined before, and $Prob\,[pos = j|pat = i]$ the probability that the newest instruction in this pattern is in slot j in the decode stage. The inequality $j \geq d_U(i)$ indicates that we only need to account penalty when the contending ALU instruction is in the ID-stage at the same cycle. Note we also defined the function $\mathrm{fr}(i)$, since we will use it in the next sections.

To illustrate the formula, consider again pattern "XAAA" and assume it was part of the instruction stream "XXXXAAA" (In other words: the pattern was preceded by non-ALU instruction (marked as 'X')). The last ALU instruction in this pattern can enter the ID-stage in four different positions, see the ID-stage at t_0 in the four subfigures of Figure 3.5. In case (a) A_3 enters the ID-stage at slot 3. When the two older ALU instructions start execution at t_1, A_3 gets blocked because all ALU's are occupied. Since this was the last instruction in the ID-stage at t_0 one slot will be unused in the EX-stage at t_1 (marked with 0), hence we loose $\frac{1}{4}$ of a cycle. In case (b) A_3 enters the ID-stage in slot 2 at t_0. It gets blocked from going to the EX-stage at t_1 because all ALU's are occupied, executing A_1 and A_2. Next to blocking itself it also blocks the younger instruction that was at slot 3 in the ID-stage of t_0. This means that we now loose $\frac{2}{4}$ of a cycle. In cases (c) and (d) enough ALU's are free to process the ALU instructions and hence no cycles are lost. This is because the instruction at $d_U(i) = 2$, i.e., instruction A_1, resides in a later stage and no contention occurs.

Assuming the four situations have an equal probability to occur (uniform distribution of the instructions in the pipeline), we derive the penalty for this instruction stream as follows:

$$
\begin{aligned}
c_{fu}(XAAA) &= (\mathrm{Prob}[\mathrm{pos} = 0] + \mathrm{Prob}[\mathrm{pos} = 1]) \times 0 \\
&\quad + \mathrm{Prob}[\mathrm{pos} = 2] \times \frac{2}{4} + \mathrm{Prob}[\mathrm{pos} = 3] \times \frac{1}{4} \\
&= \frac{1}{4} \times 0 + \frac{1}{4} \times 0 + \frac{1}{4} \times \frac{2}{4} + \frac{1}{4} \times \frac{1}{4} \\
&= \frac{3}{16},
\end{aligned}
$$

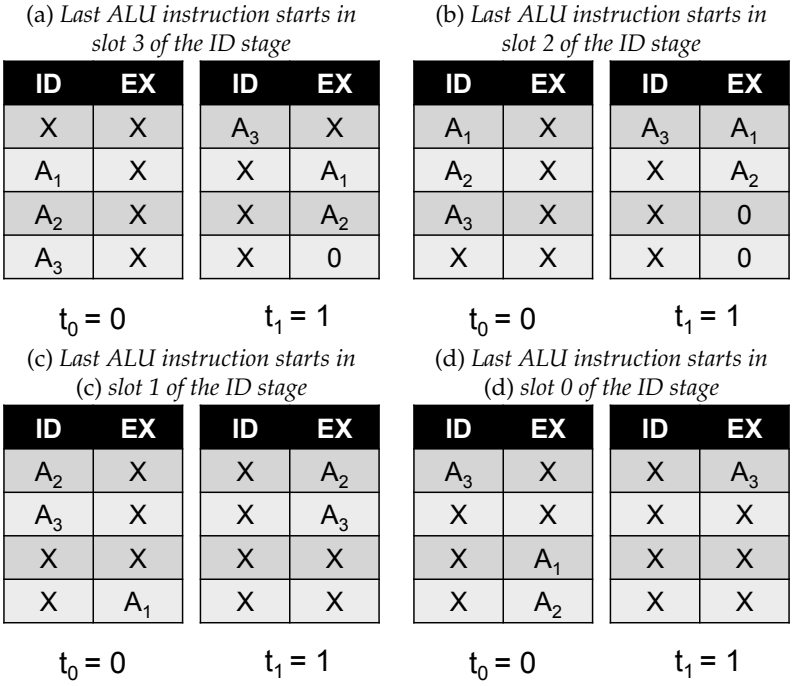


Figure 3.5: Four possible instruction flows of the pattern "XAAA"

which is exactly the same as we would become with Equation 3.22.

Special case: number of ALU instructions $\geq$ number of ALUs+2 Note that the approximation in Formula 3.22 again made use of the assumption that instructions are uniformly distributed in the pipeline stage. This assumption holds true in all cases where the number of ALU instructions in the pattern is smaller than the number of ALUs+2. However, in case we have fewer units (e.g., $U = 1$) than ALU instructions in the pattern (e.g., pattern "AAXA"), some positions will make the instruction shift to another position, because of contention between older ALU instructions in the pattern. For example, for the pattern "$A_1A_2XA_3$" and one ALU, only A_1 will be executed first, stalling A_2 for one cycle. Hence, A_3 will have to stall for two cycles. To model this effect we make a slight exception to the assumption of a uniform distribution of instructions in the pipeline stage. Instead we redistribute the probability mass function for the slots an instruction shifts between. For the example pattern, slot 3 would get assigned a probability of 0 instead of $\frac{1}{W}$, because if A_3 originally entered the ID-stage in slot 3 it gets shifted to slot 2. We therefore increase the probability of slot 2 by $\frac{1}{W}$, resulting in an effective probability of $\frac{2}{W}$ for slot 2.

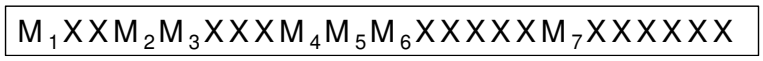


Figure 3.6: An example instruction stream with multiply instructions.

Non-pipelined functional units with long latencies

We now derive formulas for long-latency instructions for a fixed number of functional units. We first explain how to model non-pipelined units, and then discuss pipelined units in the next section. Although the formulas are general enough to be applied to all types of functional units, we focus on integer multiply instructions[4] to derive them. The only parameter that needs to be adjusted for other types of functional units is the latency.

Accounting penalty for a limited number of integer multiply instructions can be split up in two parts: (1) the fraction of cycles that is lost because of multiple M-instructions in the same stage in the same cycle; and (2) the additional cycles we need to wait for the previous multiply instruction to finish (because they are not pipelined). The first part can be calculated as before using Formula 3.21. The second (and largest) part requires knowledge of how many multiply instructions can be issued in parallel.

We start with an example instruction stream that can be found in Figure 3.6. To ease the discussion we introduce subscripts to number the multiply instructions. Figure 3.7(a) shows four snapshots of the execution of this instruction stream, displaying the state of the EX-stage and MEM-stage, according to detailed simulation for a processor with two multiply units, where the execution latency of a multiply instruction is 5 cycles. The snapshots are chosen so that they reflect the start of each group of multiply instructions that can issue in parallel. For example, the first two multiply instructions (i.e., M_1 and M_2) start execution in the EX-stage in cycle 0. In cycle 1 (i.e., t_0) they flow to the MEM-stage and stay there for 4 cycles, leaving the MEM-stage for the WB-stage at cycle 5. This makes the multiply units available to start execution at cycle 5 for instruction M_3. At cycle 6, instruction M_4 can start execution in parallel with instruction M_3. The absolute start cycles are of less importance here. More important is to note that we distinguish 4 groups of multiply instructions, meaning that we penalize the performance by 4 times the multiply latency for the 7 'M'-instructions.

The challenge now is to calculate the penalty while having limited in-

[4] In practice, modern integer multiply units are typically pipelined; older processors such as the Alpha 21064 and MIPS R4000 have non-pipelined interger multiply units. We merely use the integer multiply instruction as an example throughout the paper to explain the construction of the formulas.

(a) *Instruction flow for non-pipelined functional units*

EX	MEM
0	M_1
0	X
0	X
0	M_2

$t_0 = 1$

EX	MEM
M_4	M_3
0	X
0	X
0	X

$t_1 = 6$

EX	MEM
X	M_5
X	M_6
X	X
0	X

$t_2 = 12$

EX	MEM
X	X
X	X
X	M_7
X	X

$t_3 = 16$

(b) *Instruction flow for pipelined functional units*

EX	MEM
M_3	M_1
X	X
X	X
X	M_2

$t_0 = 1$

EX	MEM
M_6	M_4
X	M_5
X	0
X	0

$t_1 = 6$

EX	MEM
X	X
X	X
X	M_7
X	X

$t_2 = 11$

Figure 3.7: Instruction flow of the example instruction stream in Figure 3.6 through a superscalar processor with two multiply units.

Pattern	Frequency
XXXM	3
MXXM	1
XXMM	2
XMMM	1
XXMX	2
XMXX	2
XMMX	1
MMMX	1
MMXX	2
MXXX	3
XXXX	5

Table 3.1: H-matrix for the instruction stream in Figure 3.6.

formation compared to detailed simulation, i.e., using only the pattern distribution matrix in Table 3.1. With Formula 3.21, we can calculate $\mathrm{fr}(i)$, the fraction of cycles lost because there are more multiply instructions than multiply units available at the same time in the same stage. To determine the parallelism of multiply instructions that can get executed at the same time we can again use the pattern distribution matrix. We consider patterns that end with an M-symbol and account an appropriate penalty, based on

Pattern	Penalty
XXXM	$\text{latency} - 1$
MXXM	0
XXMM	0
XMMM	$\text{latency} - 1$

Table 3.2: Penalties for the patterns of the H-matrix in Table 3.1 in the case of two non-pipelined multiply units.

the total number of M-symbols in the pattern.

For the example where we have two multiply units, we need to account the full multiplier latency of 5 cycles when the rightmost multiply instruction is the only multiply instruction in the pattern, because it will block the MEM-stage during its execution. When the rightmost instruction in the pattern is the second multiply instruction, it can be issued in parallel with the previous one, hence no penalty needs to be accounted for. If the rightmost multiply instruction is the third multiply instruction, both multiply units are busy executing the first two, thus we need to account another 5 cycles. A fourth multiply instruction can again be executed in parallel, accounting for zero penalty. We summarize the penalties for the pattern distribution matrix of Table 3.1 in Table 3.2. We can now determine the total penalty for the example by using the $fr(i)$ function to account for bubble instructions as before, and by multiplying the penalties of the second column of Table 3.2 with their associated distribution from Table 3.1:

$$P_{\text{FU}} = \text{fr}(i) + 4 \times (\text{latency} - 1) + 3 \times 0$$

In general we can derive the following Formula to calculate the total cost for executing a long-latency instruction:

$$c_{fu}(i) = \text{fr}(i) + \begin{cases} \text{latency} - 1 & \text{if } (\#\text{insns}(i) \bmod U) = 1 \\ 0 & \text{else} \end{cases} \tag{3.23}$$

In this formula U is the number of units of this type of long-latency instruction, and #insns(i) represents the number of instructions of this type in pattern i. The intuition behind this formula is that the first multiply instruction will appear as a "XXXM" pattern, for which we account the full penalty. The second multiply instruction will see the first instruction as an older instruction in its pattern (e.g., XMXM), so we do not need to account a penalty if there are two or more multiply units.

Special case: dense concentrations of long latency instructions. Formula 3.23 can lead to underestimations in situations where we have a

dense concentration of multiply instructions. As an example, consider the instruction stream "XXXMMMMMMMMMMMM" and 4 multiply units. This instruction stream would be accounted for only 1 full penalty of (latency-1) instead of 3, because only the pattern "XXXM" introduces a penalty. This can be solved by modifying the previous formula to:

$$c_{fu}(i) = \text{fr}(i) + (\text{latency}-1) \times \begin{cases} 1 & \text{if } (\#\text{insts}(i) \bmod U) = 1 \\ \frac{Pr[\text{dense}|\text{pattern}=i]}{\min(U, \#\text{insts}(i))} & \text{else} \end{cases} \tag{3.24}$$

Here, $Pr[\text{dense}|\text{pattern} = i]$ is the probability that pattern i appears in a dense concentration of multiply instructions. More specifically we need to account penalty for those patterns of which none of the instructions were ever accounted a full penalty of latency-1. We can do this by calculating probabilities from the H-matrix. As an example consider the pattern "$M_1M_2M_3M_4$" and four available functional units. If M_1 was preceded by three other multiply instructions, no penalty was ever accounted to M_1. If on the other hand M_1 was preceded by three X-instructions a full penalty was accounted to M_1 and hence, the 3 younger multiply instructions can execute in parallel. In order to be able to determine which of those situations happened, we need to estimate by which instructions the current pattern was preceded. The probability that "$M_1M_2M_3M_4$" was preceded by 3 X-instructions can be calculated with the following Formula:

$$\begin{aligned} Pr[XXXM_1|M_1M_2M_3M_4] = \quad & Pr[XXXM_1|XXM_1M_2] \\ & \times Pr[XXM_1M_2|XM_1M_2M_3] \\ & \times Pr[XM_1M_2M_3|M_1M_2M_3M_4] \end{aligned} \tag{3.25}$$

In Formula 3.25 we have decomposed the probability that estimates the 3 previous instructions into probabilities that estimate the probability of 1 previous instruction. Estimating the probability of a single previous instruction can be derived from the H-matrix. Consider the following example:

$$Pr[XM_1M_2M_3|M_1M_2M_3M_4] = \frac{Pr[XM_1M_2M_3]}{Pr[XM_1M_2M_3] + Pr[MM_1M_2M_3]}$$

In the case of four multiply units $Pr[\text{dense}|\text{pattern} = MMMM] = (1 - Pr[XXXM|MMMM])$ as 3 X-instructions in front of the first multiply is the only pattern for which a full penalty could have been accounted. Different patterns and different number of functional units need to calculate

Pattern	Penalty
XXXM	$\text{latency} - 1$
MXXM	0
XXMM	0
XMMM	0

Table 3.3: Penalties for the patterns of the H-matrix in Table 3.1 in the case of two pipelined multiply units.

the probability of different preceding patterns. Calculating these probabilities can be done in a fully automated way and we find it to improve the accuracy of the our model in several cases.

Pipelined functional units with long latencies

We now move to modeling the impact of long-latency instructions that execute on pipelined functional units. Pipelined functional units have the advantage that they are capable of executing a new instruction every cycle, hence they yield a potential performance improvement over non-pipelined execution, but they also require a more complex design. Because of in-order execution there is however a limit on the potential performance improvement: when a long-latency instruction is being executed it will block all younger instructions from passing the MEM-stage (because of in-order commit). This will make the EX-stage fill up with instructions, blocking younger instructions from starting execution. So, only instructions that can make it into the EX-stage can potentially execute in parallel.

In Figure 3.7(b) we illustrate what happens with the example in Figure 3.6 if the multiply units are pipelined. We distinguish three groups of multiply instructions that can be executed in parallel (assuming no dependences between them).

As before, we can use the distribution matrix in Table 3.1. To calculate the penalties we again account for two parts: the part in which we lose a fraction of a cycle because there are more multiply instructions at the same time in the EX-stage than multiply units available; and the part where we need to account for the latency of the multiplication itself. For pipelined units, we only need to account for this latency if the current multiply instruction is the only multiply in the pattern. Table 3.3 summarizes the penalties for pipelined units. Using these penalties and the distribution matrix we can account for the total penalty caused by long-latency instructions in this example instruction stream:

$$P_{\text{FU}} \quad = \quad \text{fr}(i) + 3 \times (\text{latency} - 1)$$

In general, the penalty for long-latency instructions on pipelined functional units can be calculated with Formula 3.26. We only account a penalty if there is exactly one multiply instruction in the pattern, because all other instruction can be executed in parallel. As is the case with non-pipelined functional units, we also account for a penalty in case there is a high density of multiply instructions.

$$c_{fu}(i) = \mathrm{fr}(i,U) + \begin{cases} \mathrm{latency} - 1 & \text{if } \#\mathrm{insts}(i) = 1 \\ \frac{\mathrm{latency}-1}{\#\mathrm{insts}(i)} \times P[\mathrm{dense}|\mathrm{pattern} = i] & \text{else} \end{cases} \tag{3.26}$$

3.5 Experimental setup

We use 19 benchmarks from the MiBench benchmark suite [28], which is a popular suite of embedded benchmarks from different application domains, including automotive/industrial, consumer, office, network, security, and telecom, see also Table 3.4. Next to these MiBench benchmarks, we also use 15 benchmarks from SPEC CPU 2006, which can be found in Table 3.5. Although the benchmarks of SPEC CPU 2006 are more server-oriented benchmarks rather than embedded benchmarks, they serve as a good validation of the model as they tend to stretch the microarchitecture in different regions than MiBench. Furtheremore, superscalar in-order processors could form an interesting alternative to out-of-order processors in servers, because of their energy-efficiency. We selected inputs from the KDataSets input database [10], so that each MiBench benchmark executes approximately 1 billion instructions. For SPEC CPU 2006, we used representative simulation regions of 1 billion instructions each using SimPoint [30]. The selection of SPEC CPU 2006 benchmarks is based on the SimPoints that were available in the lab and compatible with the simulation framework.

Figures 3.8 and 3.9 illustrate our framework for evaluation using detailed cycle-level simulation and evaluation using the proposed analytical model, respectively. We use an extended version of the superscalar in-order model of the gem5 simulator [3] as a detailed cycle-level simulator to validate our mechanistic model and derive our profiler from gem5's functional simulator. Detailed simulation runs at 92 KIPS on an Intel Xeon Harpertown (L5420) processor. Our profiler is more than ten times faster, running at 1.4 MIPS, and moreover, profiles need be calculated only once for a whole range of processor configurations. With these profiles we can quickly generate performance estimates by evaluating the analytic formulas in the previous sections. This is done in a couple of seconds for the complete design space. This means that, for the experiments performed in this chapter, estimating performance through the model was done in

Benchmark	*Category*	*Description*
adpcm_c	Telecommunication	Pulse code modulation with ADPCM (encode)
adpcm_d	Telecommunication	Pulse code modulation with ADPCM (decode)
dijkstra	Network	Shortest path calculation between nodes in a graph
gsm	Telecommunication	Voice encoding with GSM
jpeg_c	Consumer	Lossy image compression with JPEG standard
jpeg_d	Consumer	Decompression of JPEG compressed images
lame	Consumer	MP3 encoding
patricia	Network	Compression of network data structures
qsort	Automobile/Industrial	Data sorting (e.g., 3D coordinates)
rsynth	Office	Text to speech synthesis
sha	Security	Secure hashing
stringsearch	Office	Searching of words in phrases (case insensitive)
susan_c	Automobile/Industrial	Corner recognition in images
susan_e	Automobile/Industrial	Edge recognition in images
susan_s	Automobile/Industrial	Image smoothening
tiff2bw	Consumer	Conversion of a TIFF image to black and white
tiff2rgba	Consumer	Conversion of a TIFF image into RGB formatted TIFF
tiffdither	Consumer	Dithering a black and white TIFF image
tiffmedian	Consumer	Reducing the color palette of an image

Table 3.4: Overview of MiBench benchmarks

less than 8 hours, while estimating through detailed simulations took 752 computing-days (spread out over a cluster of computers).

We use McPAT [46] for our power estimates. The inputs for McPAT are various processor configuration parameters, such as pipeline depth, width, cache configuration, memory latency, chip technology, etc.; along with pro-

Benchmark	*Category*	*Description*
bwaves	Fluid Dynamics	Computes 3D transonic transient laminar viscous flow
bzip2	Compression	Julian Seward's bzip2 version 1.0.3
cactusADM	Physics/General Relativity	Solves the Einstein evolution equations
gemsfdtd	Computational Electromagnetics	Solves the Maxwell equations in 3D using the FDTD method
gobmk	Artificial Intelligence	Plays the game of Go
gcc	C Compiler	Based on gcc 3.2, generates code for Opteron
hmmer	Search Gene Sequence	Protein sequence analysis using Hidden Markov Models
h264	Video Compression	A reference implementation of H264/AVC
lbm	Fluid Dynamics	Simulates incompressible fluids in 3D
libquantum	Physics/Quantum Computing	Simulates a quantum computer, running Shor's factorization algorithm
mcf	Combinatorial Optimization	Vehicle scheduling using a network simplex algorithm
omnetpp	Discrete Event Simulation	Models a large Ethernet campus network
perlbench	Programming Language	Derived from Perl V5.8.7
povray	Image Ray-tracing	Image rendering
sjeng	Artificial Intelligence	A highly-ranked chess program
xalancbmk	XML Processing	Transforms XML documents to other document types
zeusmp	Physics/CFD	Simulation of astrophysical phenomena

Table 3.5: Overview of SPEC CPU 2006 benchmarks

gram parameters, such as number of instructions, instruction mix, etc.; and finally, program-machine parameters, such as cache misses, branch mispredictions, etc. Note that, when detailed cycle-level simulation is used as input to McPAT, detailed statistics can be gathered directly from the simulation output, see Figure 3.8. When using the mechanistic model, fewer

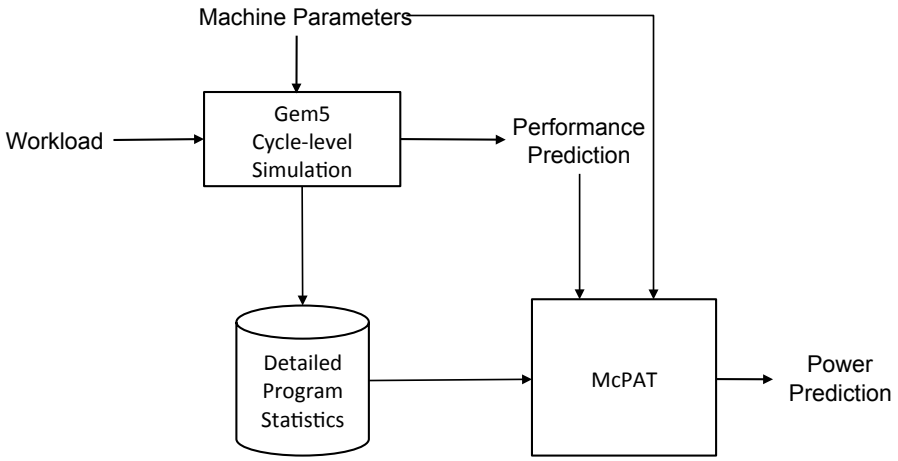


Figure 3.8: Simulation framework when evaluation is driven by cycle-level simulation.

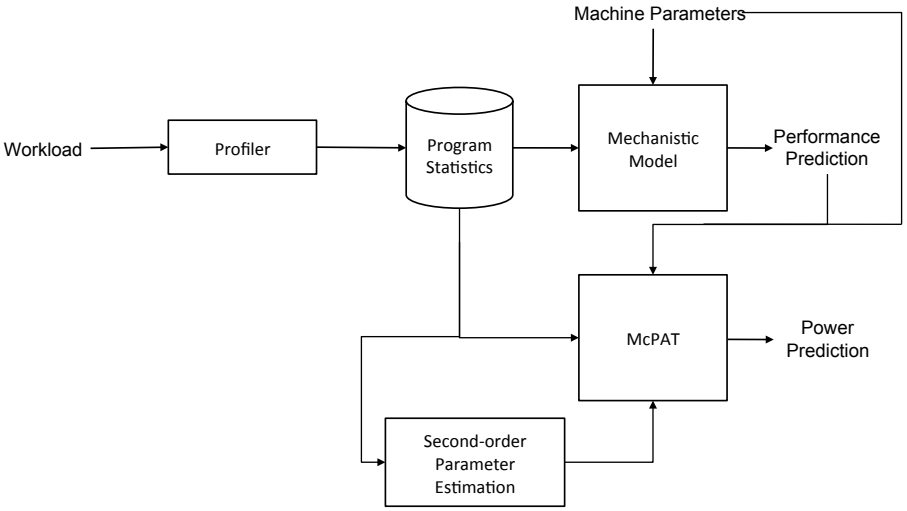


Figure 3.9: Simulation framework when evaluation is driven by the mechanistic model.

information is available and hence some second-order parameters need to be estimated, as illustrated in Figure 3.9. We find these parameters to have a much less significant impact on performance than the information available in the profile, and are able to make accurate power predictions, as can be observed in the experiments of Section 3.7.2.

Throughout the next section we will be alternating through two design spaces α and β as shown in Tables 3.6 and 3.7 respectively. We use Design Space α to vary a number of key core- and uncore- parameters, while Design Space β will be used to focus on the functional units only. Using

Parameter	*Baseline* α	*Range* α
I-cache	32KB 4 way set-assoc 64 byte blocks	32KB 4 way set-assoc 64 byte blocks
D-cache	32 KB 4 way set-assoc 64 byte blocks	32KB 4 way set-assoc 64 byte blocks
L2-cache	512KB 8 way set-assoc 10ns latency	128KB – 256KB – 512KB – 1MB 8 vs 16 way set-assoc 10ns latency
pipeline depth	9 stages 1GHz	5 – 7 – 9 stages 600MHz – 800MHz – 1GHz
processor width (W)	4 slots	1 – 2 – 3 – 4 slots
branch predictor	1KB global history	1KB global history – 3.5KB hybrid 10b local and 12b global history
Unified ALU	4 units 1 cycle latency	W units 1 cycle latency
Unified Multiply/ Divide Unit	1 unit non-pipelined 5 cycles multiply latency 20 cycles divide latency	1 unit non-pipelined 5 cycles multiply latency 20 cycles divide latency

Table 3.6: Design Space α

two design spaces allows us to both explore the broad applicability of the model and have a fine-grained evaluation on the most complex part of it. The total number of evaluated data points (design spaces α and β over the considered workloads) equals 6,028.

The default processor configuration in Design Space α is a a superscalar in-order processor with private 32KB L1 caches and a unified L2 cache, as can be seen in Table 3.6. Further, we also vary a number of important microarchitecture parameters in this design space, such as pipeline depth and frequency setting (3 configurations), pipeline width (4 configurations), L2 cache size and associativity (8 configurations), as well as branch predictor configuration (2 configurations). This leads to a design space consisting of 192 design points within which we will be evaluating the model's accuracy. Although this is not a very large space compared to real design spaces, it was close to the limit we could explore given our infrastructure because we compare model accuracy against detailed simulation results which are very time-consuming and costly to obtain — which is the motivation for this research in the first place. Note that the number of ALU's here are scaled along with the superscalar width W. The number of Multiply/Divide units equals one. This reduces the complexity of the formula's for functional unit contention and hence allows us to focus on the rest of the microarchitecture.

Parameter	*Baseline β*	*Range β*
I-cache	128KB 4 way set-assoc 64 byte blocks	128KB 4 way set-assoc 64 byte blocks
D-cache	128KB 4 way set-assoc 64 byte blocks	128KB 4 way set-assoc 64 byte blocks
L2-cache	4MB 8 way set-assoc 10ns latency	4MB 8 way set-assoc 10ns latency
pipeline depth	5 stages 1GHz	5 stages 1GHz
processor width	4 slots	4 slots
branch predictor	1KB global history	1KB global history
Integer ALU (IA)	2 units 1 cycle latency	1 to 4 units 1 cycle latency
Integer Multiply/ Divide Unit (IM)	1 unit non-pipelined 5 cycles multiply latency 20 cycles divide latency	1 to 4 units non-pipelined/pipelined 5 cycles multiply latency 20 cycles divide latency
Floating-Point ALU (FA)	1 unit non-pipelined 3 cycles latency	1 to 4 units non-pipelined/pipelined 3 cycles latency
Floating-Point Multiply/ Divide Units (FM)	1 unit non-pipelined 15 cycles latency	1 to 4 units non-pipelined/pipelined 15 cycles latency

Table 3.7: Design Space β

The parameters that we vary in Design Space β are shown in Table 3.7, along with the default settings, which forms a design space of 2048 configurations. The primary focus of this design space is on the processor core in which we vary the number and the configuration of the functional units, as this is the most complicated part to model for superscalar in-order processors. This also explains the choice of using larger caches: in order to carefully evaluate the functional unit contention model, having the memory subsystem as a bottleneck must be avoided. To validate the functional unit contention model, we use a subset of 70 randomly selected configurations that span a broad range of the design space of the total 2048 configurations.

For the hardware validation we use a BeagleBone Black board with the Texas Instruments AM3358 Sitara ARM Cortex-A8 processor [5] , running Debian GNU/Linux 7. The corresponding microarchitectural parameters can be found in Table 3.8 and are based on the technical references from ARM [33] and Texas Instruments [35]. Because the Cortex-A8 is an ARM-processor we extended our profiler to capture profiles for the ARM ISA, in addition to the ALPHA ISA, which was used for the validation against de-

[5]We chose the Cortex-A8 because of its popularity in embededded devices such as some of the Samsung Galaxy smartphones and tablets and in the Apple iPad.

Parameter	*Cortex-A8*
I-cache	32KB 4 way set-assoc 64 byte blocks
D-cache	32KB 4 way set-assoc 64 byte blocks
L2-cache	256KB 8 way set-assoc
pipeline depth	13 stages 1GHz
processor width	2 slots
Integer ALU (IA)	2 units 1 cycle latency
Integer Multiply Unit (IM)	1 unit pipelined 3 cycles multiply latency
Floating-Point Unit (FU)	1 unit non-pipelined 10 cycles arithmetic latency 10 cycles logic latency 17 cycles multiply latency 65 cycles divide latency 26 cycles MAC latency 60 cycles sqrt latency

Table 3.8: Cortex-A8 microarchitectural parameters

tailed simulation. We used Linux' built-in time command to measure user time and we disable dynamic frequency scaling by setting the processor to a fixed clock frequency of 1GHz during our experiments.

3.6 Model Validation

We now analyze the accuracy of the model by comparing the performance predictions of the model with those of detailed simulation through a series of experiments. In addition, we compare the performance predictions of the model to real hardware performance on the ARM Cortex-A8.

3.6.1 Validation Against Detailed Simulation

We validate the model in three steps. First, we evaluate how well the model tracks the relative difference when varying the number of functional units: starting from Baseline β, we increase the number of available functional units. We also study the effect of pipelining functional units. Second, we evaluate the model against a broad range of different configurations within design spaces α and β to get an overall estimate of the accuracy. Third,

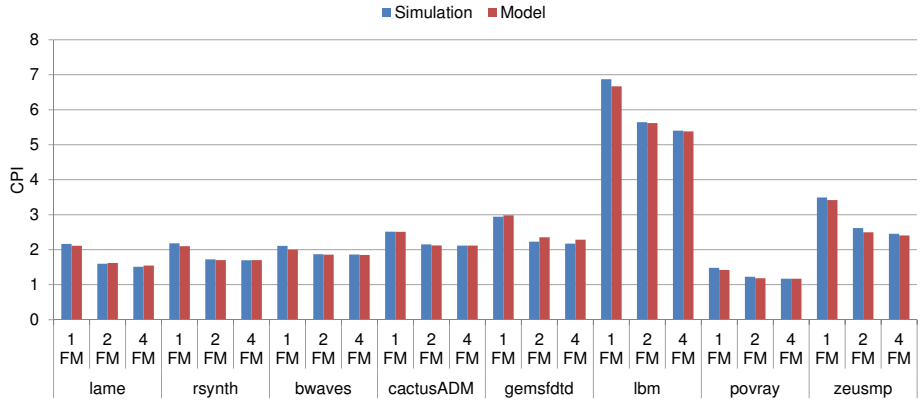


Figure 3.10: Model validation while varying the number of floating-point multiply units (FM), for the floating-point benchmarks of MiBench and SPEC CPU2006.

we evaluate how well the model tracks whether a benchmark scales with superscalar width.

Varying the functional units We start by studying the effect of adding floating-point multiply units to the baseline configuration of Design Space β. As can be seen in Figure 3.10, adding a second multiply unit decreases CPI significantly. Four multiply units reduces CPI only for a few benchmarks. These trends are tracked accurately by our model. On average, the model has an error of only 2.1% for the configurations in this experiment and a maximum error of 5.5%.

We now look at the impact of pipelining all the units (i.e., integer multiply/divide unit, floating point ALU and floating point multiply/divide unit) of the baseline configuration on the SPEC CPU 2006 benchmark suite, as shown in Figures 3.11 and 3.12. Again our model predicts the performance difference accurately, with an average absolute error of 2% and a maximum error of 7.6%.

Design space exploration We now evaluate accuracy by comparing the model with detailed simulation for a large range of different configurations. We compare the CPI values of detailed simulation versus the model for both Design Space α and β. For Design Space α we evaluate the 19 Mibench benchmarks of Table 3.4 and a full design space exploration on all 192 points. For Design Space β we consider the 34 benchmarks from Tables 3.4 and 3.5 and 70 microarchitectural configurations. Figure 3.13 shows cumulative plots comparing the simulated points with the model. The golden curve shows the cumulative plot over all simulated points. From this Figure we can see that about 90% of all evaluated points have an error of less than 6.2% in CPI. Overall, for the 6,028 data points we have considered, the

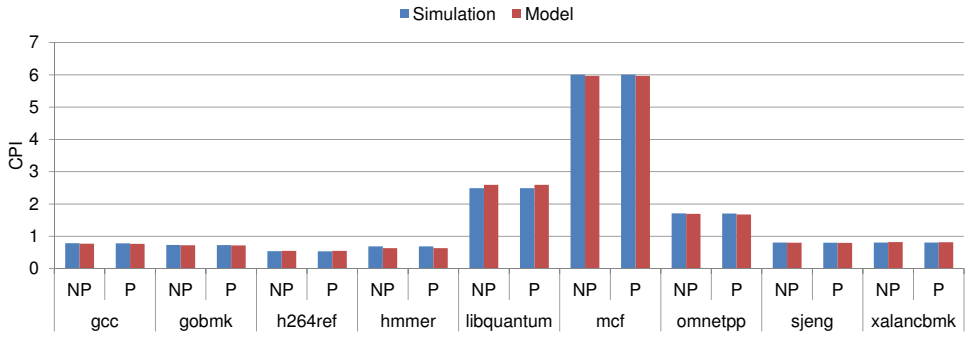


Figure 3.11: Evaluation of the model compared to detailed simulation for pipelined (P) and non-pipelined (NP) functional units, on the integer benchmarks of SPEC CPU 2006.

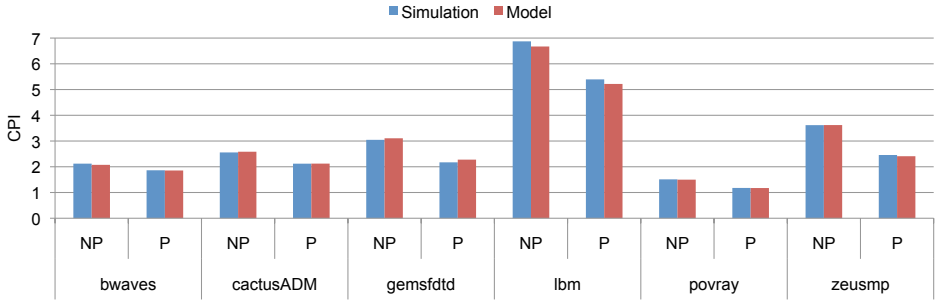


Figure 3.12: Evaluation of the model compared to detailed simulation for pipelined (P) and non-pipelined (NP) functional units, on the floating-point benchmarks of SPEC CPU 2006.

model has an error of 2.8% on average with a standard deviation of 0.024, and a maximum error of of 13.1%.

We also report cumulative plots for Design Spaces α and β individually. For Design Space α we observe that 90% of all evaluated points have an error of less than 5.5% in CPI. For Design Space β we observe that 90% of all evaluated points have an error of less than 7%. For Design Space α the model has an average error of 2.5% with a standard deviation of 0.020, while for Design Space β the average error is 3.2% with a standard deviation of 0.028. Note that the slightly larger error for Design Space β is caused by the more complex design points in its design space, such as the large superscalar width and the non-unit latencies for floating point ALU instructions.

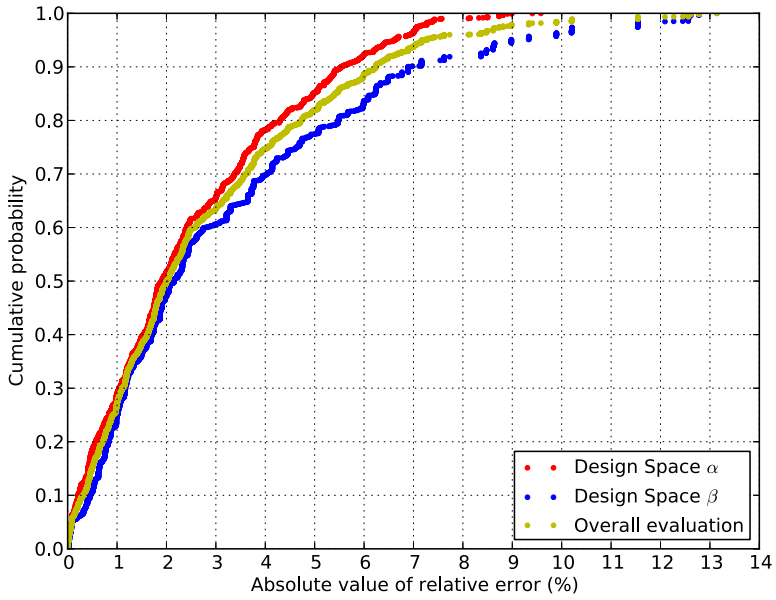


Figure 3.13: Cumulative probability distribution of error for design spaces α and β on all evaluated points.

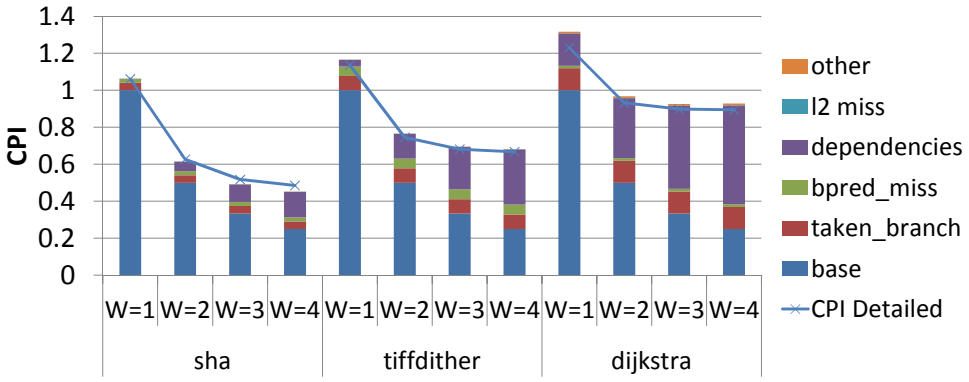


Figure 3.14: Model accuracy for estimating relative performance as a function of superscalar width.

Varying superscalar width. Figure 3.14 shows CPI stacks [6] as obtained through the model, as a function of superscalar width. The overall CPI obtained through detailed simulation is also shown as a reference. The three benchmarks were picked based on how they scale with processor width. The sha benchmark benefits the most from superscalar processing, whereas dijkstra benefits the least; tiffdither is somewhere in the middle. This graph in fact demonstrates the amount of insight that mechanistic modeling of-

[6] Section 3.8.1 explains how these CPI stacks can be obtained. We show the stacks here to understand how some benchmarks scale with superscalar width and others don't.

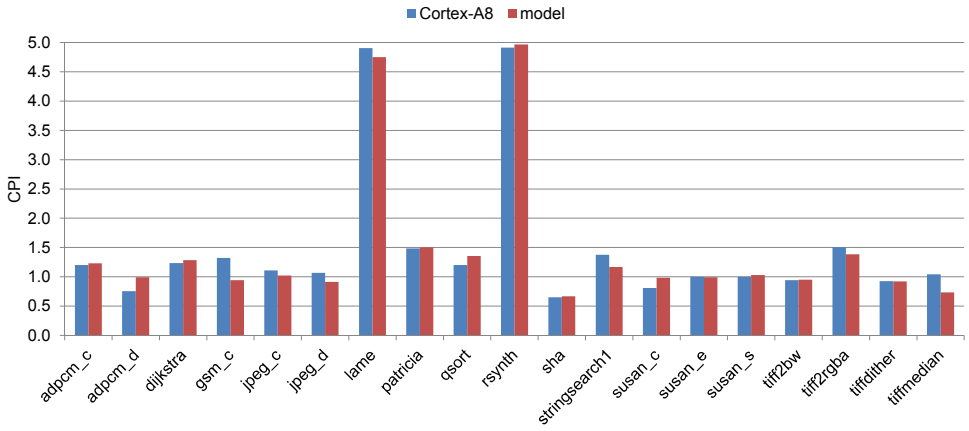


Figure 3.15: Evaluation of the model compared to the Cortex-A8 microarchitecture.

fers beyond detailed simulation, because it breaks up overall performance into its contributing factors. Clearly, although there is a benefit from going from scalar processing to 2-wide processing for dijkstra, going beyond 2-wide processing does not improve performance much. The reason why is immediately clear from the CPI stacks: although the base component (N/W) decreases, its decrease is compensated by more inter-instruction dependences, which impede the benchmark from benefiting from superscalar processing. This is not the case for sha which seems to suffer less from dependences; apparently, this benchmark exhibits more ILP (Instruction Level Parallelism).

3.6.2 Validation Against Hardware

Figure 3.15 shows CPI-values for our set of 19 MiBench benchmarks when executed on the Cortex-A8 processor, along with the prediction of our model. We find that the average absolute prediction error is 10% with a standard deviation of 0.10. For 12 benchmarks we find an error of less than 8%. The maximum error is 32% for adpcm_d. Overall the model is fairly accurate, taken into account that we did not make important changes to the model compared to the ALPHA/gem5 model. The only changes are adjustments to the profiler to be compatible with the ARM ISA, and modeling Cortex A8's variable latencies for floating point instructions.

More specifically, to improve the accuracy of the floating point benchmarks we required a more fine-grained breakdown of the floating point instruction mix (i.e., multiply, division, multiply-accumulate and square root). We then use this fine-grained instruction mix together with instruction latencies found in the Cortex-A8 technical reference [33], to generate a weighted average of the overall floating point latency and feed it into our

model.

Several adjustments to the model could be made to improve accuracy even more. Gem5 models an fetch buffer with the size of an entire cache line, as observed by Gutierrez et al. [29], which underestimates the number of instruction cache accesses in gem5. Because our original modeling efforts are targeted at a microprocessor similar to gem5's in-order core, we make the same assumptions on the fetch buffer. gsm_c for example is an application within the top 3 benchmarks with most instruction cache misses, indicating that its instruction footprint is non-cache-resident, and hence modeling a smaller fetch buffer would correctly predict a higher execution time. Furthermore we are unaware of the branch prediction latency of the Cortex-A8, while we model taken branches with 1 cycle of penalty. In addition, Gutierrez et al. [29] show that gem5's branch prediction accuracy decreases for low MPKI values. Both these branch predictor inaccuracies could explain our performance underestimation of adpcm_d. Further, the Cortex-A8 does not allow executing two instructions in the same slot if they have output dependences (WAW dependences), while our model abstracts this away. This likely impacts the overall performance overestimation. Finally, we assume a fixed memory latency while DRAM latency tends to be dependent on the physical memory addresses requests are made for. As correctly stated by Desikan et al. [14], this depends on the virtual to physical page mappings of the native system and is very difficult to replicate. We find that benchmarks that spend a lot of time in system calls, such as tiff2rgba and tiffmedian, indeed show performance overestimations.

3.7 Guiding design space exploration

3.7.1 Minimizing Number of Functional Units for a Given Performance Target

In this case study, we use the model to minimize the number of units needed for a specific performance target. We use gem5 to find the performance (expressed as IPC) of Baseline β (containing 5 functional units) and the maximum achievable performance when having four functional units of each type (i.e., a total of 16 units). The harmonic mean of speedups over the baseline IPC equals 1.087. The maximum speedup is observed for lame (1.52) and zeusmp (1.49).

We now use the model to find optimal configurations per benchmark (i.e., configurations with a minimum number of functional units), where we set the performance target at 98% of the maximum IPC. Detailed simulation of these optimized configurations confirms that these configurations indeed have an IPC of at least 98% of the maximum achievable IPC as

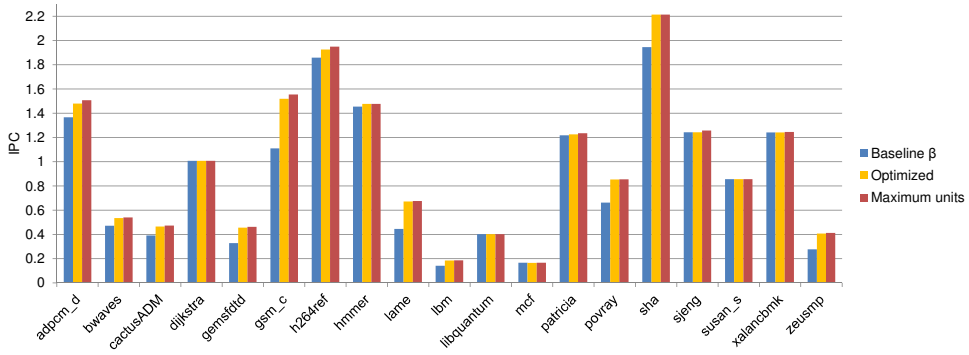


Figure 3.16: Baseline performance, performance of the configuration with 4 units of each type (Maximum units) and the performance of the configuration picked by the model with a minimum number of functional units within 98% of the optimum (Optimized).

predicted by the model, by using a minimum amount of functional units. The harmonic mean of the speedups of the optimized configurations over Baseline β equals 1.08 (maximum speedup of 1.51 for lame and 1.47 for zeusmp). Table 3.9 shows the benchmark-optimal configurations found using the model, for all the benchmarks in our setup. The table illustrates that, for 32 benchmarks we only need 7 or less functional units to achieve at least 98% of the IPC with 16 functional units. These configurations are not trivial and time-consuming to find using detailed cycle-accurate simulation, which motivates the use of our fast model to guide design choices. For a selection of benchmarks we show IPC numbers, resulting from detailed simulation, in Figure 3.16. The "Optimized" bars, represent the IPC results for the configurations in Table 3.9. The bars "Baseline" and "Maximum units" represent our baseline of 5 units, and the configuration with all 16 units, respectively. The selection is a mix of benchmarks that both show large speedups and very low speedups when increasing the number of functional units.

3.7.2 Minimizing the Energy Delay Product

Processor designers take various metrics into account during the development process. Energy consumption clearly is a key metric when designing embedded processors. We now explore design space β (see Table 3.7) while considering both performance and energy consumption. We therefore consider the energy-delay product (EDP) which is defined as the product of energy consumption and execution time. When optimizing for EDP, designers are mostly interested in the design point with the lowest EDP. Therefore it is appealing to be able to use the model to guide the search for the point of minimal EDP.

Units	IA	IM	FA	FM	benchmarks
4	1	1	1	1	mcf
5	2	1	1	1	libquantum, perlbench, bzip2, dijkstra, xalancbmk, gcc, sjeng, omnetpp
6	1	1	2	2	cactusADM
6	2	1	1	2	qsort
6	2	1	2	1	gobmk
6	2	2	1	1	patricia
6	3	1	1	1	tiffdither, susan_c, jpeg_c, tiffmedian, hmmer, adpcm_d, h264ref, stringsearch1 adpcm_c, tiff2rgba, susan_s
7	1	1	2	3	gemsfdtd
7	2	1	2	2	rsynth, bwaves
7	3	2	1	1	susan_e, jpeg_d, tiff2bw
7	4	1	1	1	sha
8	2	1	2	3	povray
8	4	2	1	1	gsm_c
9	1	1	4	3	lbm
9	2	1	2	4	zeusmp
11	2	1	4	4	lame

Table 3.9: Benchmark-optimal configurations achieving at least 98% of maximum performance with a minimum number of units.

To limit the number of detailed simulations, we have conducted two experiments. We first consider the full design space exploration of non-pipelined functional units of Table 3.7 for four benchmarks in Figure 3.17. Next we explore a smaller subspace for all the other benchmarks in Figure 3.19.

In Figure 3.17 we compare EDP obtained through detailed simulation against the mechanistic model; we use McPAT, as mentioned before, for the power modeling. For these four benchmarks we performed 256 detailed simulations (i.e., all possible combinations with non-pipelined units in Table 3.7). The detailed simulations serve as input for McPAT, as illustrated earlier in Figure 3.8. Next we use the model to generate input for McPAT for the exact same configurations. Apart from providing an estimate of the CPI, we also modeled second-order statistics (such as idle time, unit occupancies, etc.) needed by McPAT, as illustrated in Figure 3.9. On the X-axis we sorted the configurations from lowest to highest EDP according to detailed simulation. The Y-axis shows EDP as found by detailed simulation versus EDP found by the model. As can be seen in Figure 3.17 the model follows detailed simulation very closely.

For the other benchmarks in our setup, we now use detailed simula-

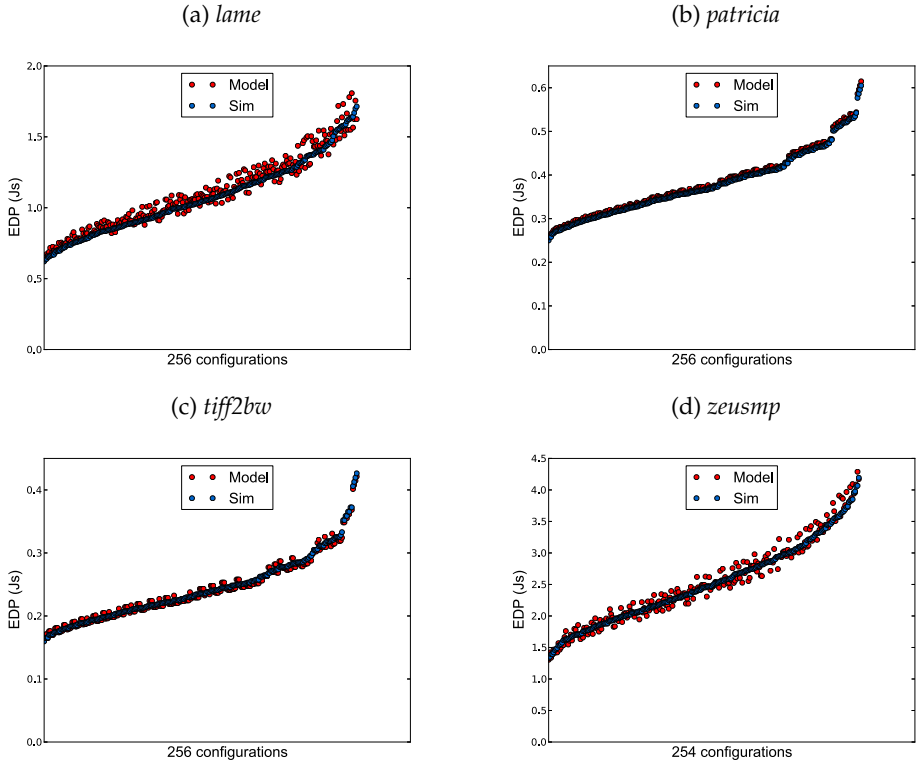


Figure 3.17: Using the model versus detailed simulation when optimizing for EDP, for four benchmarks and 256 configurations.

tion with McPAT on a smaller subspace of the complete design space. As illustrated in Figure 3.18, we first use the model to select a number of microarchitectural configurations for each benchmark that have low predicted EDP values. More specifically, we select configurations so that all design points within 13% (i.e., the maximum error we noticed in the evaluation in Section 3.6) of the configuration with the predicted minimum EDP are covered. We enforce a minimum of 20 configurations per benchmark. The maximum number of configurations using this selection procedure is 36.

Figure 3.19 shows EDP values as reported by detailed simulation, normalized against the EDP of Baseline β. We plot the "Model Optimum", i.e., the simulated EDP value for the optimal design point as predicted by the model, and the "Simulated Optimum", i.e., the EDP value of the optimum design point found by detailed simulation. When we compare the optimum predicted by the model with the optimum predicted by detailed simulation, we observe only few differences: for most benchmarks the optima have the same EDP, and on average we have an EDP difference of 1%. The maximum EDP difference is 6% for adpcm_c. Compared to the baseline architecture, the model helps us to reduce 6% in EDP on average, and

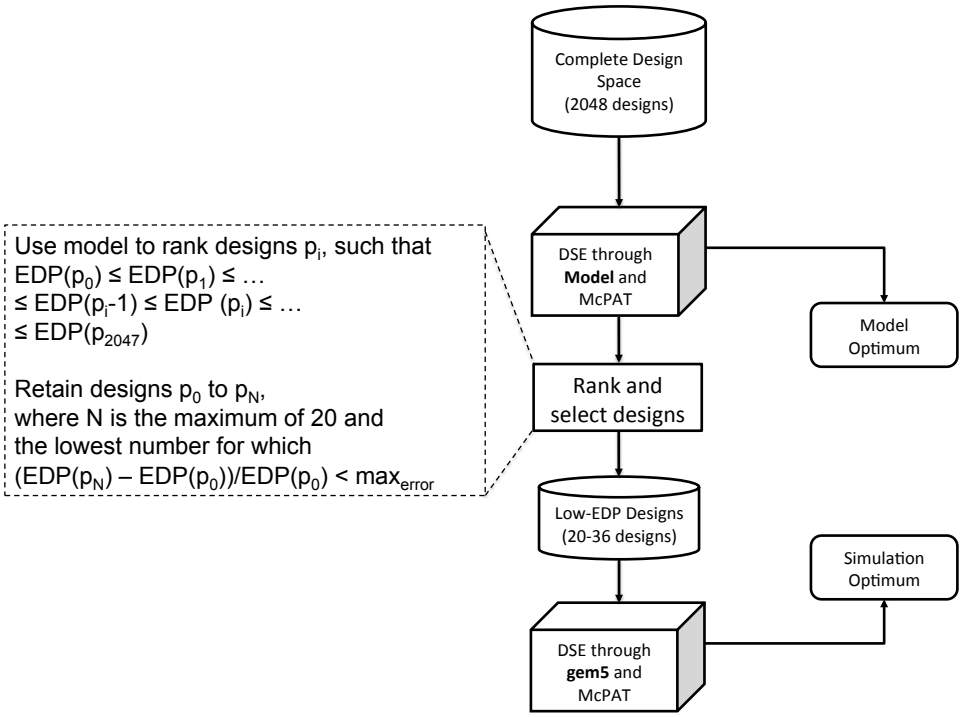


Figure 3.18: Framework to filter the designs used in the second experiment of Section 3.7.2, to limit the number of designs to evaluate with detailed cycle-level simulation.

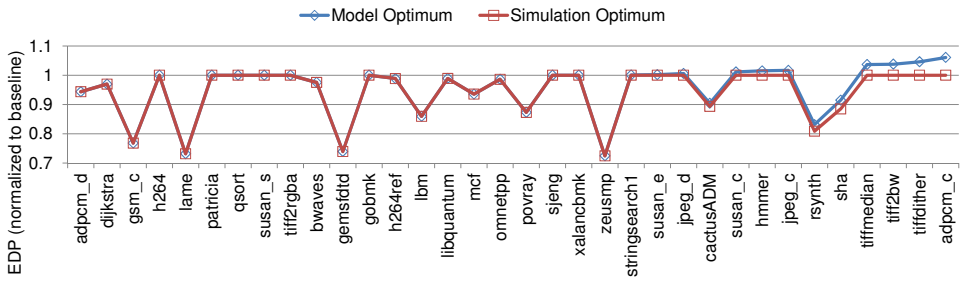


Figure 3.19: EDP (normalized by the EDP of Baseline β) for the lowest EDP configuration discovered by simulation and by the model.

up to 28%. Using detailed simulation, the average EDP reduction is 7% and up to 28%. Note that for a large number of the benchmarks, the baseline configuration has a normalized EDP close to 1, indicating that the baseline happens to be optimal. This does not mean that their EDP is constant as for these benchmarks we observe variations in EDP (up to a factor of 2.5). This can also be seen by looking at the wide EDP ranges for the benchmarks patricia and tiff2bw in Figure 3.17.

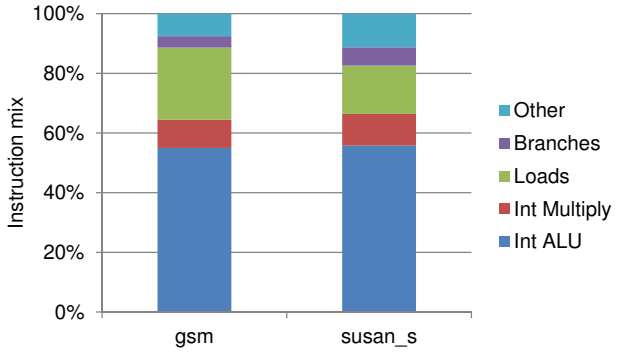


Figure 3.20: The instruction mix of benchmarks gsm_c and susan_s are similar: Many integer ALU instructions and over 10% integer multiply instructions.

3.8 Gaining insights

3.8.1 Revealing Performance Bottlenecks

This case study illustrates the additional benefit of the mechanistic performance model over empirical models: the ability to break down the total number of executed cycles into smaller terms that provide a better level of detail into how the application interacts with the microarchitecture.

While well-trained empirical models provide a significant speedup in simulation time compared to cycle-level simulators and are able to show the performance impacts of different processor designs, it is challenging at times to reveal the underlying reason why a design improves performance for one application but not another. We illustrate this with an example: Figure 3.20 shows two benchmarks with a very similar instruction mix, namely gsm_c and susan_s. Considering the baseline configuration of design space β, we would expect that adding integer ALUs or integer multiply units would improve performance, based on the instruction mix. Although there appear fewer multiply instructions in the instruction mix than ALU instructions, multiply instructions are more expensive since they have a longer execution latency. We therefore simulate the baseline configuration and a configuration in which we have two multiply units instead of one and plot the CPI (Cycles Per Instruction) for both configurations. Figure 3.21 shows that gsm_c indeed experiences a significant performance improvement (the CPI is significantly lower). However, despite a similar instruction mix, susan_s' behavior is completely different: we barely observe a performance improvement when adding the second multiply unit. In this section we show how the mechanistic model provides a level of insight that enables quick and deep understanding of such performance phenomena.

Formula 3.1 derives the number of execution cycles of an application on

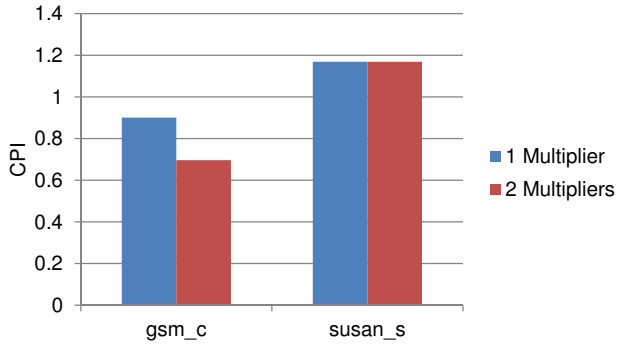


Figure 3.21: Adding an additional multiply unit increases performance significantly for one benchmark, but not for the other, while the instruction mixes of Figure 3.20 are similar.

a target microprocessor as a sum of terms. This property is very useful in determining performance bottlenecks. By identifying the largest contributors to the execution cycles, one can find the most promising directions to improve performance. For example, suppose the penalty due to data cache misses (part of the P_{misses} term) is the largest contributor to the execution cycles, performance could be improved by installing a larger cache, or by improving data locality. If, on the other hand the penalty due to functional units (P_{FU}) is relatively large, we can improve performance by adding functional units.

We now build CPI stacks for the benchmarks and configurations of Figure 3.21. Thereto, we divide the terms in Formula 3.1 by the total number of instructions N, and we split up the terms P_{misses} and P_{FU} into smaller terms to get a better level of detail. The 'base' component is the first term, i.e., $\frac{N}{W}$, which becomes $\frac{1}{W}$ when divided by N. P_{misses} can easily be split up by multiplying the number of miss events of each type with their respective penalty as explained in Section 3.3.

Since P_{FU} and P_{deps} are modeled in a unified matrix C, we have to split up the matrix C into C_{FU} and C_{deps} to determine penalties for functional units and inter-instruction dependences separately. We do this similarly to how the matrix was constructed in Formula 3.7:

$$C_{deps}(i,d) = \begin{cases} c_{dep}(i,d) & \text{if} c_{dep}(i,d) > c_{fu}(i) \\ 0 & \text{else} \end{cases}$$

$$C_{FU}(i,d) = \begin{cases} c_{fu}(i,d) & \text{if} c_{dep}(i,d) \leq c_{fu}(i) \\ 0 & \text{else} \end{cases}$$

The term P_{FU} can be further broken down into P_{intALU}, $P_{intMultiply}$, P_{fpALU} and $P_{fpMultiply}$ by accounting for the $C_{FU}(i,d)$ terms only, in which i denotes a pattern associated with that functional unit (i.e., the last instruction in pattern i executes on functional unit type FU).

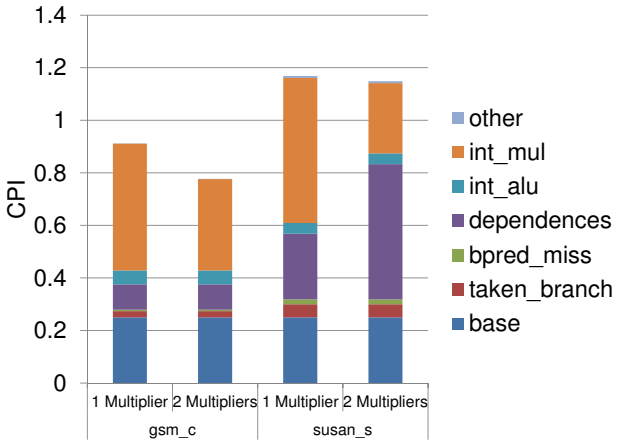


Figure 3.22: CPI stacks reveal that inter-instruction dependences between multiply instructions are the underlying bottleneck that is preventing performance improvement for susan_s. The 'other' component are all other terms in the model that only have a small component.

The CPI stacks for the benchmarks and configurations of Figure 3.21 are shown in Figure 3.22. Although the model has a small error on the performance prediction, it reveals why there is hardly any performance improvement for susan_s. With a single multiply unit, we observe that the penalties due to integer multiplies ($P_{intMultiply}$) is relatively high. When we add a second multiply unit, we see that this term is significantly reduced for gsm_c, which can be explained by the many patterns with more than one multiply instruction. For susan_s, we see that the $P_{intMultiply}$ term is also reduced, however the term P_{deps} is increased by the same amount $P_{intMultiply}$ was reduced. The decrease in $P_{intMultiply}$ can again be explained by patterns consisting of multiple multiply instructions that can execute in parallel. However, the increase in P_{deps} means that these multiply instructions depend on each other, which inhibits parallel execution.

3.8.2 Compiler Optimizations

We now use the model to study how compiler optimizations affect superscalar in-order performance, see Figure 3.23. We consider `-O3`, `-O3` without instruction scheduling (`-O3 -fno-schedule-insns`), and `-O3` with loop unrolling turned on (`-O3 -funroll-loops`) for the five benchmarks for which we observed the largest impact due to compiler optimizations. Figure 3.23 shows normalized cycle stacks, i.e., a cycle stack is computed by multiplying a CPI stack with the number of dynamically executed instructions; the cycle stacks are then normalized to the execution time with the `-O3` optimization level. For most of the benchmarks, instruc-

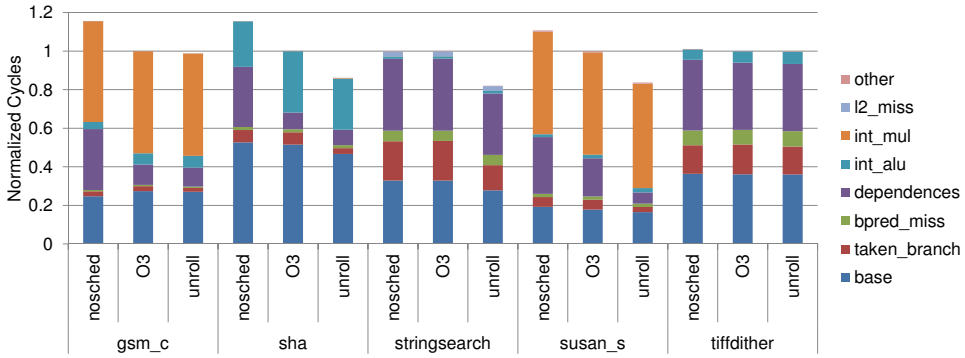


Figure 3.23: Normalized cycle stacks for five benchmarks across different compiler optimizations.

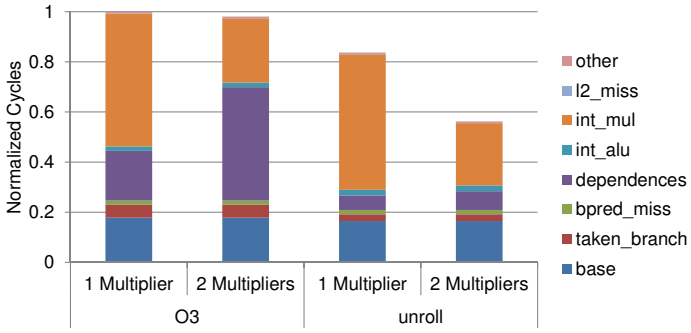


Figure 3.24: Normalized cycle stacks for susan_s with and without loop unrolling, and on two different architectures (one and two multiply units).

tion scheduling increases the distance between dependent instructions, resulting in a lower penalty due to dependences. For some benchmarks, e.g., gsm_c, the base component increases slightly through instruction scheduling, meaning that the number of executed instructions increases. The reason for this is the addition of spill code. However, the cost of spill code is compensated for by the substantial decrease in the impact of inter-instruction dependences.

Most of the benchmarks (and all the ones in Figure 3.23) benefit from loop unrolling. Three components get an important reduction through loop unrolling. First, the number of dynamic instructions decreases because fewer branches and loop iteration counter increments are needed after loop unrolling. Second, because there are fewer branches, the penalty due to taken branches also decreases. The third and biggest contribution comes from the smaller penalty due to inter-instruction dependences; clearly, loop unrolling enables the instruction scheduler to better schedule instructions so that fewer inter-instruction dependences have an impact on in-order performance.

Configuration Parameter	*Value*
I-cache	32KB 4 way set-assoc 64 byte blocks
D-cache	32 KB 4 way set-assoc 64 byte blocks
L2-cache	512KB 8 way set-assoc 10ns latency
pipeline depth	5 stages 1GHz
processor width	4 slots
branch predictor	1KB global history
Integer ALU (IA)	2 units 1 cycle latency
Integer Multiply/ Divide Unit (IM)	1 unit 5 cycles multiply latency 20 cycles divide latency
Floating-Point ALU (FA)	1 unit 3 cycles latency
Floating-Point Multiply/ Divide Units (FM)	1 unit 15 cycles latency

Table 3.10: Configuration used to compare in-order with out-of-order CPI stacks.

In particular, for susan_s, loop unrolling substantially reduces the impact of dependences. As shown in Section 3.8.1, dependences prevent a performance improvement when the number of multipliers is increased from one to two for susan_s. In Figure 3.24, we show normalized cycle stacks for susan_s without loop unrolling (`-O3`) and with loop unrolling enabled (`-unroll`) on the baseline architecture of Design Space β (1 Multiplier) and on the baseline architecture with an additional multiply unit (2 Multipliers). As mentioned earlier, for `-O3` the penalty for multiply instructions transforms into a penalty for inter-instruction dependences when a second multiply unit is added. When we enable loop unrolling, however, many of these additional inter-instruction dependences can be removed because the instruction scheduler is now able to place independent multiply instructions (that were originally spread across loop iterations) closer together and dependent ones further apart. As a result, we see a considerable performance gain when the number of multiply units is doubled for the loop-unrolled version of susan_s.

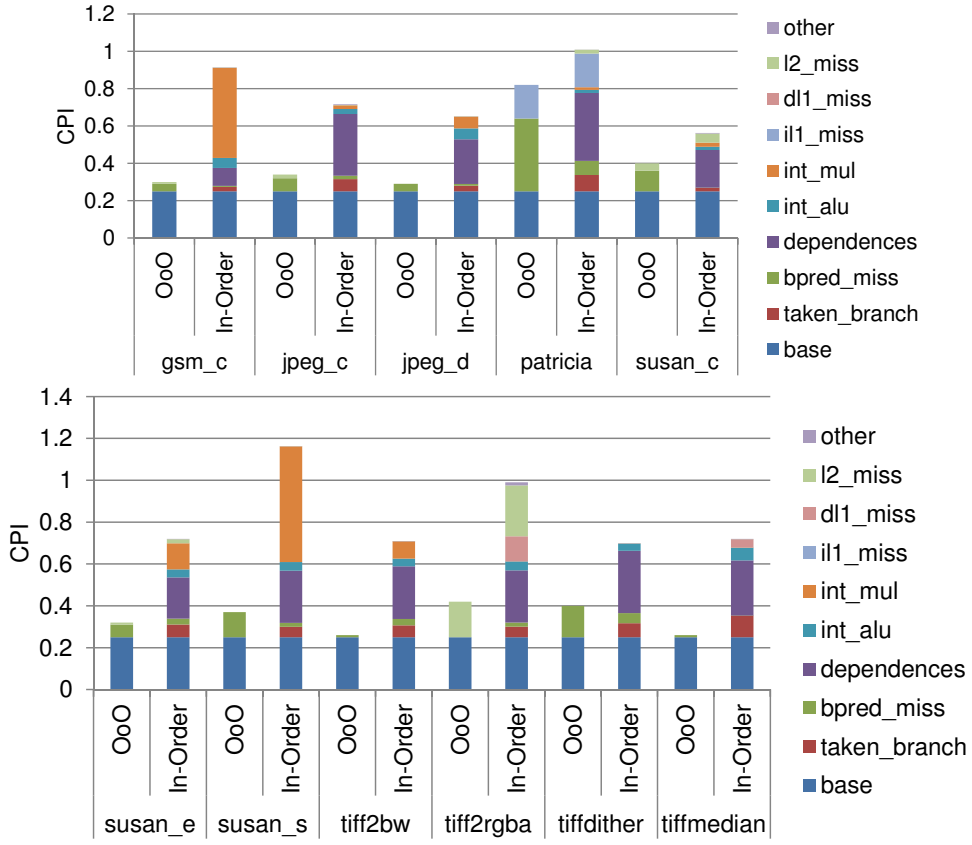


Figure 3.25: Comparing in-order versus out-of-order performance using CPI stacks obtained through mechanistic modeling.

3.8.3 In-order versus out-of-order performance

In this application, we compare in-order versus out-of-order performance using CPI stacks, see Figure 3.25 [7]; We only show CPI stacks for a selected number of benchmarks to improve readability. The in-order CPI stacks are obtained using the model described in this chapter; the out-of-order CPI stacks are obtained using the model described in prior work [24]. In this comparison, we consider four-wide in-order and out-of-order processors with parameters configured as in Table 3.10. A number of fundamental and insightful observations can be made from this graph.

- Dependencies are largely hidden by out-of-order execution, in contrast to in-order processing. This is apparent for all the benchmarks.

[7] The CPI stacks were obtained by reimplementing the model in the SimpleScalar toolset. The use of SimpleScalar, along with the use of another cross compiler, is the reason for the slightly different results for the in-order model compared with the results in the other sections. SimpleScalar expects COFF binaries, unlike gem5 which reads the ELF format.

- Non-unit instruction execution latencies due to multiply/divide operations have significant impact on performance on in-order processors for some benchmarks, such as gsm_c and susan_s. Non-unit latencies are mostly hidden by out-of-order execution.

- Most of the benchmarks suffer from a small but noticeable penalty due to insufficient ALU's on in-order processors. While the width of the processor is four, the processor sometimes gets stalled because only two ALU's are available. Out-of-order processors on the other hand are capable of scheduling other instructions until the ALU's are available.

- The cost per mispredicted branch is larger on out-of-order processors than on in-order processors, see for example patricia. The reason is that on an in-order processor, the cost equals the depth of the front-end pipeline, whereas on an out-of-order processor the branch resolution time (the time between the branch getting dispatched and executed after all instructions it depends on are finished) also contributes to the overall penalty in addition to the front-end pipeline.

- The L2 cache component is smaller on the out-of-order processor compared to the in-order processor, see for example tiff2rgba. The reason is that an out-of-order processor can better exploit memory-level parallelism and issue independent loads and stores to memory simultaneously. An in-order processor on the other hand would stall on the first use of a load miss, preventing subsequent (independent) load misses to go to memory.

- Since I-cache miss penalty is a function of the miss latency only, the penalty is identical on in-order and out-of-order processors.

This case study clearly illustrates the insight that can be obtained from mechanistic analytical modeling, which is much harder to obtain through detailed cycle-accurate simulation.

3.9 Summary

In this chapter we propose a performance model for superscalar in-order processors that uses analytical formulas derived from understanding the internal mechanics of the microarchitecture. The formulas are based on how a superscalar processor interacts with its functional units and how inter-instruction dependences through registers block the processor from sustaining a high throughput of executed instructions. By combining a detailed instruction mix and dependence distance profiles of a program's

execution with a number of program-machine characteristics (e.g., cache miss rates, MLP and branch misprediction rates), we demonstrate that our model has an error of only 2.8% on average compared to detailed simulation with gem5. Further, we are able to use our model to estimate the performance of the ARM Cortex-A8 with an average absolute error of 10%. The evaluation speed of the model is close to instantaneous, as it only involves solving a number of analytical formulas. Furthermore, the microarchitectural independent profiling step, needed to provide the model input, is a one-time cost and is at least 10 times faster than a single detailed simulation run.

We use the model both as an exploration tool and as a tool to get insight into an application's execution behavior and to visualize microarchitectural bottlenecks. We demonstrate how the model can find an optimal set of functional units to achieve a given performance target. When we combine the model with McPAT to calculate the energy-delay-product (EDP) of a given application on a given microarchitecture, we can find a design within 1% on average of the optimal EDP compared to detailed simulation. For most benchmarks, we find the same optimum with the model as with simulation. Next, we demonstrate the model's usefulness to identify microarchitectural bottlenecks. Instead of analyzing results of many detailed simulations, the model can visualize how an application interacts with a microarchitecture and hence provides insights on how performance can or cannot be improved. By applying this visualization technique on differently optimized binaries of the same application, the model provides insight into how compiler optimizations impact the program-microarchitecture interactions. Finally we compared in-order versus out-of-order performance by generating CPI-stacks with the respective mechanistic models.

Chapter 4

Selecting Representative Benchmark Inputs for Design Space Exploration

The fact that we live at the bottom of a deep gravity well, on the surface of a gas covered planet going around a nuclear fireball 90 million miles away and think this to be normal is obviously some indication of how skewed our perspective tends to be.
Douglas Adams

There are basically two ways for reducing the number of cycle-level simulations during design space exploration: the number of microarchitectural configurations can be reduced or the number and/or size of workloads can be reduced. The mechanistic model described in the previous chapter can help designers reduce the number of configurations. For reducing the workloads, prior research in workload analysis has focused on benchmark selection and finding small, but representative samples within a benchmark application, as described in Chapter 2. Very little attention, however, has been given to selecting representative inputs for these benchmark applications. Instead, common practice is to assume that different inputs lead to similar run-time behavior and hence no prior selection is done.

While the behavior may be relatively insensitive to the inputs for some applications, other applications may be very sensitive. For example, a streaming application, such as a filter operation, may be largely input-insensitive as the application executes the same code and accesses memory in a highly predictable way no matter what input it is given. On the other hand, a video application that decodes an action movie with lots of complex scenery versus a recording of a news reader with little variation across

subsequent frames, is likely to lead to different code regions being executed, as well as different memory access patterns being observed, which in its turn leads to different run-time behavior.

In this chapter, we use a large-scale experiment to quantify how skewed our design decisions can potentially be by relying on inputs that are unrepresentative for the application. We report average EDP increases of 57% and 33% when using one and three badly selected inputs, respectively. While this indicates that careful input selection is important for designing general-purpose processors, it is even more important for application-specific processors that target specific application domains.

Drastically increasing the number of simulations to cover as much application behavior as possible is infeasible, given the low simulation speeds. Therefore, this chapter proposes and evaluates three different methods that identify *representative* inputs, at very low one-time costs and as few as three inputs.

This chapter is organized as follows. Section 4.1 illustrates the potential loss of not using representative inputs with a case study. Section 4.2 details the experimental setup and derives the performance metric that will be used throughout the rest of the chapter. Section 4.3 provides a detailed study on the impact of implicit workload parameters on design space exploration. Section 4.4 introduces our three novel techniques to select representative inputs and compares the trade-offs between the techniques in Subsection 4.4.5. Finally, Section 4.5 summarizes the chapter.

We also made the identification of representative inputs open to the community, by creating a Bitbucket project [1] , see also Appendix B.

4.1 Potential pitfall of current practice

We first set up a limited experiment to further motivate the problem and gain some initial insight, before diving into a systematic evaluation of the importance of benchmark inputs during design space exploration. We consider one benchmark from the MiBench benchmark suite [28], namely sha, a secure hash algorithm, with five inputs and five processor configurations. The five inputs were randomly selected from the 1,000 input sets provided through KDataSets [10], and the five processor configurations are shown in Table 4.1; we consider superscalar in-order processors and vary issue width, data and instruction cache size, branch predictor configuration and pipeline depth. We simulate these processor configurations using the gem5 simulator [3].

[1] At the time of writing, this project is located at `https://bitbucket.org/maximilien_breughe/representative-benchmark-inputs/wiki/Home`

	conf0	conf1	conf2	conf3	conf4
Issue width	2	4	4	2	4
Data cache size	16 KB	64 KB	8 KB	64 KB	64 KB
Instruction cache size	16 KB	8 KB	32 KB	16 KB	32 KB
Branch predictor	1 KB global	1 KB global	3.5 KB hybrid	1 KB global	3.5 KB hybrid
Pipeline depth	9	9	9	7	9

Table 4.1: Design space for detailed simulation.

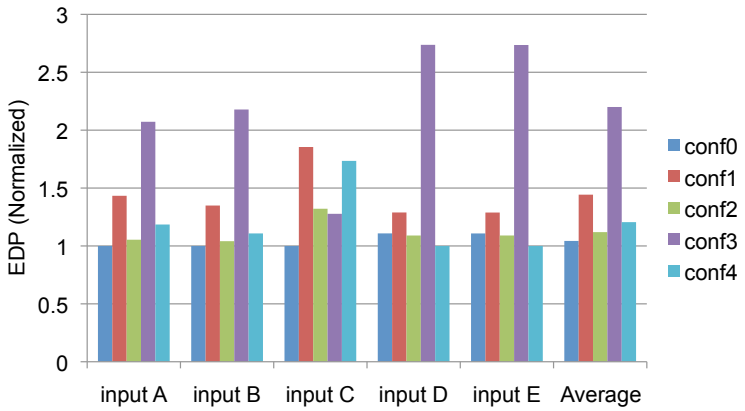


Figure 4.1: Normalized EDP for sha for five different processor configurations and five different inputs.

Figure 4.1 reports normalized energy-delay product (EDP) for each of the five inputs across these five processor configurations, along with the average EDP across all inputs. The EDP values are normalized against the processor that is optimal for the given input. Hence, an EDP value of one denotes the optimal processor configuration for a given input (e.g., configuration 0 for input C), and the closer the normalized EDP values to one, the better. It is immediately clear from the results shown in Figure 4.1 that configuration 0 is the most optimal processor configuration (i.e., the one with the lowest EDP) across all five inputs: for most inputs its EDP is close to 1 and it has the lowest EDP on average, see the 'average' bars in the graph. The same result would be obtained in case inputs A, B or C would have been chosen to guide the design space exploration. However, using inputs D or E during design space exploration would have led us to believe that configuration 4 is most optimal. The risk now is that if configuration 4 would be deployed in the field, an end user using another input, for example input C, would experience a 73% worse EDP compared to configuration 0. The pitfall here is that it is unclear at design time whether we are in the case of inputs D or E, or one of the other inputs. In other words, using a

Parameter	*Range*	*Baseline*
Issue width	1 - 2 - 3 - 4	2
Data cache size	8 KB - 16 KB - 64 KB	16 KB
Data cache associativity	2 - 4	4
Instruction cache size	8 KB - 16 KB - 32 KB	16 KB
Instruction cache associativity	2 - 4	4
Cache block size	32 - 64 byte	32 byte
Branch predictor	1 KB global - 3.5 KB hybrid	3.5 KB hybrid
Pipeline Depth	5 - 7 - 9	9

Table 4.2: Design space considered in this study.

non-representative input to drive the design space exploration may lead to a design point with poor performance for other inputs, and this is hard to know a priori (if at all possible) without doing a full exploration.

This case study motivates the need for a methodology for identifying representative inputs for design space exploration. The remainder of this chapter will further investigate and quantify the pitfall of non-representative inputs in a systematic way, and will present methods for identifying representative inputs so as to minimize the chance of ending up with a design point during design space exploration that could lead to suboptimal and even poor performance for unseen inputs when deployed in the field.

4.2 Experimental Setup

We now give a detailed overview of the design space and the workloads considered throughout this chapter. We also describe our modeling infrastructure to efficiently explore the huge design space, as well as the optimization criterion we are targeting.

4.2.1 Design Space

The design space considered in our exploration is shown in Table 4.2. We assume a superscalar in-order processor core and we vary instruction and data cache size, associativity and line size, pipeline depth, issue width, and the branch predictor configuration. We consider 2 to 4 options along each of these axes. The cross-product of all these design options leads to a design space consisting of 1,728 microarchitectures. This is a fairly small de-

sign space compared to real-life design spaces, yet it enables us to illustrate our contribution while being manageable in terms of time complexity. The baseline architecture in the rightmost column will be used for the experiment in Section 4.3.3.

4.2.2 Workloads

Table 4.3 provides an overview of the benchmarks used in this chapter. All but one of the benchmarks are taken from MiBench [28]. We use the MiBench benchmark suite for two reasons. First, MiBench is targeted towards embedded processors and devices, which aligns well with the goal of tuning processor architectures for a specific workload domain. Second, there exists a large set of inputs for each of these benchmarks, which we use to explore input sensitivity. The inputs were taken from the KDataSets database [10], which provides 1,000 inputs per benchmark. These inputs have a working set size that is too large to fit in the processor core's caches, i.e., the data working set is typically larger than the 64 KB data cache considered during the exploration. Using 1,000 inputs per benchmark leads to significant simulation time requirements for the experiments in this work, nevertheless a sufficiently large number of inputs is needed to quantify the impact of input selection on design space exploration. Chen et al. [10] found these inputs to be diverse based on a detailed characterization using both micro-architecture dependent and independent metrics.

In addition to these MiBench benchmarks, we also include the h264 benchmark from SPEC CPU2006 [32] as it is highly relevant for our application domain of interest, while exhibiting interesting dynamic, time-varying behavior not observed in the MiBench benchmarks. Because CPU2006's h264 is not part of the KDataSets inputs database, we had to create our own set of 1,000 video inputs. We took fifty raw video files from the public domain [67], and generated 4 to 11 different sequences for each of these videos, by selecting different begin and end points, leading to 250 different video sequences in total. For each video sequence, we generated 4 random encoding schemes, by varying 48 different parameters. Parameters include varying the number of B frames, quantization parameters for I, P and B slices, enabling/disabling pyramid encoding, etc. The end result is 1,000 video inputs for the h264 benchmark.

We compiled all of our benchmarks using the `gcc`-4.3 cross-compiler for the Alpha ISA. Our default optimization flag is `-O3`. We also evaluate the impact of compiler optimization flags on processor architecture design space exploration in one later section in this chapter. We therefore consider 250 randomly chosen combinations of compiler optimization flags. Random selection of compiler optimization flags was previously shown to give a fairly good coverage of the impact of compiler flags on overall per-

Benchmark	*Category*	*Description*
adpcm_c	Telecommunication	Pulse code modulation with ADPCM (encode)
adpcm_d	Telecommunication	Pulse code modulation with ADPCM (decode)
dijkstra	Network	Shortest path calculation between nodes in a graph
gsm	Telecommunication	Voice encoding with GSM
jpeg_c	Consumer	Lossy image compression with JPEG standard
jpeg_d	Consumer	Decompression of JPEG compressed images
lame	Consumer	MP3 encoding
patricia	Network	Compression of network data structures
qsort	Automobile/Industrial	Data sorting (e.g., 3D coordinates)
rsynth	Office	Text to speech synthesis
sha	Security	Secure hashing
stringsearch	Office	Searching of words in phrases (case insensitive)
susan_c	Automobile/Industrial	Corner recognition in images
susan_e	Automobile/Industrial	Edge recognition in images
susan_s	Automobile/Industrial	Image smoothening
tiff2bw	Consumer	Conversion of a TIFF image to black and white
tiff2rgba	Consumer	Conversion of a TIFF image into RGB formatted TIFF
tiffdither	Consumer	Dithering a black and white TIFF image
tiffmedian	Consumer	Reducing the color palette of an image
h264	Video Compression	A reference implementation of H264/AVC

Table 4.3: Overview of benchmarks

formance, and 250 combinations was found to be sufficient to achieve near optimal performance [10].

4.2.3 Modeling Infrastructure

Evaluating 20,000 workloads (i.e., 20 benchmarks with 1,000 inputs per benchmark) on 1,728 design points, results in a total of more than 34 million measurements. Therefore, using detailed cycle-level simulation (e.g., using gem5 as done in the motivation section), which typically requires several hours of simulation time per measurement, is totally infeasible as it would require several millions of compute days to perform the measurements [2].

Instead, we resort to the model that we constructed in Chapter 3 which evaluates a single measurement in the order of seconds, once the 20,000 workloads have been profiled. This allows us to collect the huge number of measurements in a reasonable amount of time (i.e., around 3 orders of magnitude faster than using detailed cycle-level simulation). The model targets superscalar in-order processors and models the performance impact of issue width, pipeline depth, non-unit instruction execution latencies, inter-instruction dependencies, cache/TLB misses and branch mispredictions. It predicts performance within 2.8% on average compared to detailed cycle-accurate simulation. The model takes as input a number of program statistics to characterize an application's instruction mix, inter-instruction dependencies, and cache and branch behavior. This collection is a one-time cost per benchmark and input. Although we need to characterize each benchmark/input pair, the characterization is much faster than detailed simulation. The other model inputs relate to the processor architecture being considered, such as pipeline depth, width, instruction latencies, etc.

We use McPAT to estimate power consumption [46]. The inputs to McPAT are various processor configuration parameters, such as pipeline depth, width, cache conguration, memory latency, chip technology (32nm), etc., along with program parameters, such as the number of dynamically executed instructions, the instruction mix, etc., and finally, program-machine parameters, such as number of cache misses, branch mispredictions, etc.

Put together, performing all experiments reported in this chapter (i.e., program profiling, evaluating the performance model and computing power estimates) required 12,000 compute days on a single machine, instead of several millions of compute days using cycle-level simulation. To collect the measurements in a reasonable amount of time, we further parallelized the experiments on a cluster of 300 machines, resulting in 40 days of simulation time.

[2] We estimate that performing the measurements with cycle-level simulation would take 3.96 million compute days, or 10,849 years of computing time on a single machine.

4.2.4 Optimization Criterion

Although the problem we want to tackle is fairly easy to state, attacking it in a systematic way is rather complex. There are multiple dimensions involved in our study: we consider multiple benchmarks, and multiple inputs per benchmark, each leading to a different dynamic instruction count; we consider multiple microarchitectures; and we consider two optimization criteria, namely performance and power/energy.

In order to accurately and confidently evaluate how well a limited set of (presumably) representative inputs, selected by our methods, captures the behavior of the complete input database, we need to have an appropriate metric. We quantify a microarchitecture's energy-efficiency using the energy-delay product (EDP), which is computed as the total energy consumed multiplied by the total execution time to perform a given unit of work (i.e., the complete execution of the benchmark with its input). Because we have multiple inputs per benchmark, we need an appropriate way of computing the average EDP across these inputs to obtain the EDP for a particular benchmark and microarchitecture. We also need a way of comparing the EDP of the presumably optimal processor determined using a limited set of inputs against the EDP of the optimal processor configurations determined using all inputs. This is done as follows:

1. Calculate the execution time $T(i,j)$ for each input i and microarchitecture j. This is done using the mechanistic analytical performance model previously described.

2. Calculate the consumed energy $E(i,j)$ for each input i and microarchitecture j. This is done using McPAT.

3. Compute the execution time $TPI(i,j)$ and energy consumption $EPI(i,j)$ per instruction. This is done by dividing execution time and energy consumption with the number of dynamically executed instructions $I(i)$, following Equations 4.1 and 4.2:

$$TPI(i,j) = \frac{T(i,j)}{I(i)}, \tag{4.1}$$

$$EPI(i,j) = \frac{E(i,j)}{I(i)}. \tag{4.2}$$

4. Identify the microarchitecture with the minimum EDP value across all inputs. EDP_{min} is computed following Equation 4.3, with N the number of inputs in the input database ($N = 1,000$ in our setup):

$$EDP_{min} = \min_j \left(\sum_i^N TPI(i,j) \times \sum_i^N EPI(i,j) \right). \tag{4.3}$$

Note this formula complies with the recommendations by [55] regarding how to compute average EDP across benchmarks.

5. We can now compute the normalized EDP ($\widetilde{EDP}$) of processor configuration j relative to the minimum EDP:

$$\widetilde{EDP}(j) = \frac{\sum_i^N TPI(i,j) \times \sum_i^N EPI(i,j)}{EDP_{min}}. \tag{4.4}$$

The metric $\widetilde{EDP}$ thus quantifies the energy-efficiency of a processor configuration relative to the optimal configuration with the lowest average EDP across all inputs and design points considered in our setup. Hence our goal is to minimize $\widetilde{EDP}$ using as few inputs as possible during the exploration. For the remainder of this chapter, we will refer to the design with minimum $\widetilde{EDP}$ as the optimal design. We will refer to $\widetilde{EDP}$ as 'normalized EDP'. Although we use this normalized EDP as our metric throughout the chapter, the proposed input selection methods are not constructed in a way that they are bound to using EDP as an evaluation metric.

4.3 Quantifying the impact of implicit parameters on microprocessor design space exploration

The main focus of this chapter is on selecting representative benchmark inputs for design space exploration, as illustrated in Figure 4.2. Therefore, we first quantify the potential impact of benchmark inputs on design space exploration in Section 4.3.1, before presenting and evaluating the input selection techniques in detail. The results show why it is important to break common practice and select inputs in a structured way.

A second implicit parameter during design space exploration is the set of compiler optimization flags used during compilation of the application. We quantify in Section 4.3.2 that compiler flags have a significantly lower impact on design decisions than benchmark inputs.

Although microarchitectural design space exploration is the primary focus of this thesis, the data that we captured for the experiments of Section 4.3.2 allows us to discuss the reciprocal problem, namely the impact of microarchitectural design choices on selecting compiler optimization flags. We discuss this reciprocal problem in Section 4.3.3, and show that the underlying microarchitecture has a significant impact on the selection of compiler optimization flags.

Note that these three quantification studies, together with the work of Chen et al. [10], provide insights on all four relationships depicted in Figure

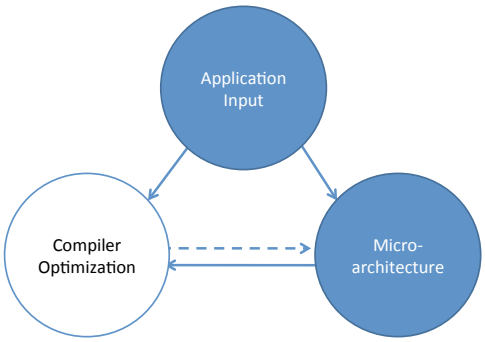


Figure 4.2: Relationships between parameters that are quantified in Section 4.3. The strong impact from benchmark inputs on microarchitectural design decisions is the main focus of this chapter, and is quantified in Section 4.3.1. In addition, the other relationships are quantified in the remainder of Section 4.3.

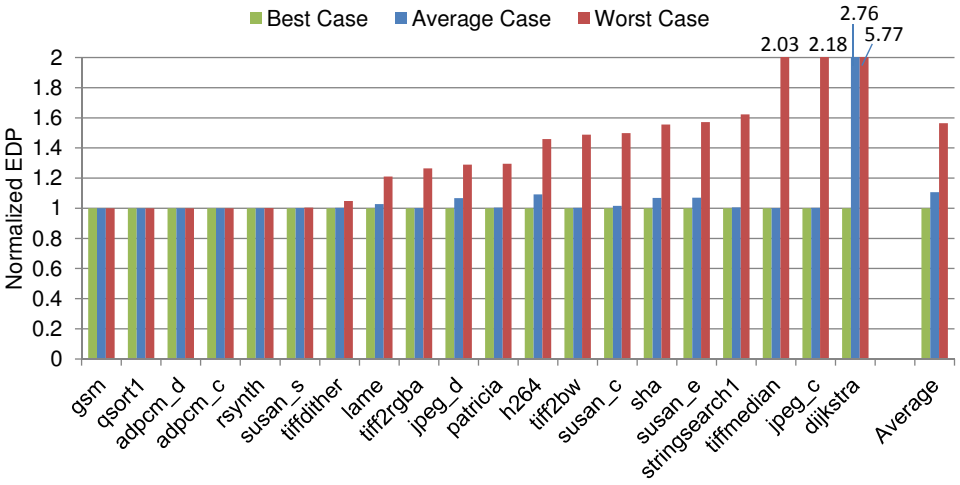


Figure 4.3: Quantifying the impact of selected benchmark inputs for identifying the optimum processor configuration.

4.2. We conclude that the selected benchmark inputs have a significant impact on microarchitectural design choices, while the impact of compiler optimization flags on design choices is much weaker. Furthermore, the choice of compiler optimization flags is strongly dependent on both microarchitectural design choices and benchmark inputs. Because of the important impact of benchmark inputs on microarchitectural design choices, we focus on input selection techniques for design space exploration in Section 4.4.

4.3.1 Sensitivity to Benchmark Inputs

We first quantify the sensitivity of benchmark inputs on design space exploration. Figure 4.3 reports normalized EDP values (as defined in Equation 4.4) across all benchmarks in our study while considering three scenarios. The first scenario ('best case') is an ideal scenario in which we would always pick a single benchmark input that, when used to determine the (presumably) most optimal processor configuration, would yield a configuration with a normalized EDP that is as close as possible to one (the normalized EDP of the globally optimal configuration across all inputs). The second scenario ('average case') is the average scenario in which we were to pick a random benchmark input for design space exploration, and reports the average EDP across these randomly selected inputs. This scenario represents what is to be expected to happen on average. The third scenario ('worst case') is the scenario in which we would be unfortunate to pick a benchmark input that would result in a (presumably) optimal processor configuration with the worst possible EDP compared to the global optimum.

There are several interesting observations to be made here. The normalized EDP values obtained through the best case scenario are equal to one for all benchmarks. This indicates that there exists at least one benchmark input that, when selected and used to drive the design space exploration, yields an optimal processor configuration that is indeed optimal across all inputs. The question now is whether it is possible to find, and if so, how to find such an input. As mentioned above, this is an ideal case, and it is hard (if not impossible) to know a priori whether a given input is going to lead to the optimal design point during design space exploration.

Picking a random input leads to a design point that is fairly competitive to the global optimum on average. For all but one of the benchmarks, we observe that a random input leads to a design point with an EDP that is close to the global optimum. For one benchmark however, namely dijkstra, a random input leads to a processor configuration with an EDP that is 176% worse on average compared to the global optimum. In other words, for some applications, selecting a single random input may lead to severe average deficiencies during design space exploration.

Picking the worst possible input leads to an EDP deficiency of 57% on average across all benchmarks, and ranges up to 477%. We observe substantial deficiencies for around two-thirds of the benchmarks; one-third of the benchmarks seems unaffected. Note these EDP deficiency numbers are average numbers across all inputs. Whereas the average EDP deficiency for a particular benchmark across all inputs can be as high as 477%, for some inputs the deficiency can be even higher, up to 82$\times$ the EDP compared to the optimum design point. This illustrates the high sensitivity of design

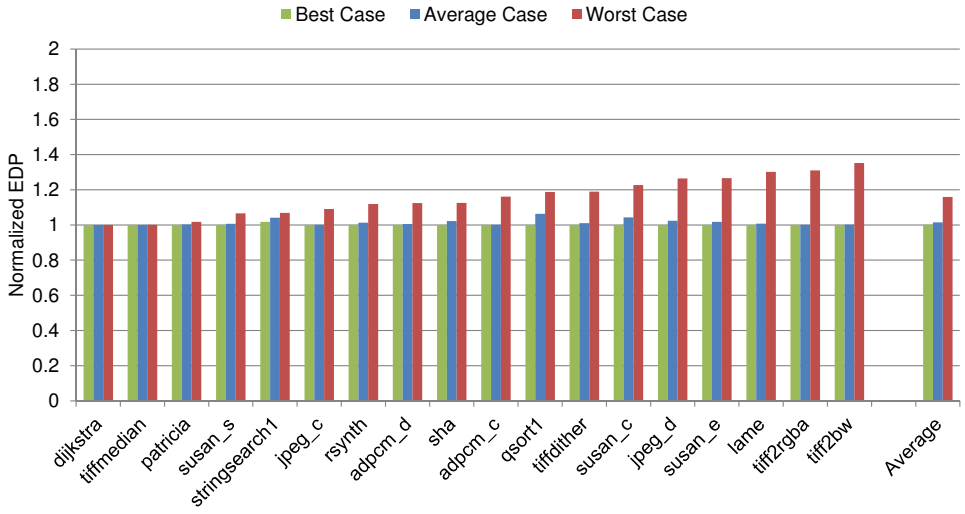


Figure 4.4: Quantifying the impact of compiler optimization flags on the identification of the optimum processor configuration.

space exploration with respect to benchmark inputs. While the worst case scenario is not the common case, or what is to be expected on average, it might happen, and the pitfall is that it is unclear a priori whether we have picked an input that represents the worst, average or best case. In other words, there is no way for a designer to verify this unless one were to explore the entire design space with all possible inputs, which is infeasible in practice. For this reason we focus on the worst case scenario. We want to avoid picking benchmark inputs that lead to suboptimal designs, and these results clearly illustrate the need for a systematic method for selecting representative benchmark inputs.

4.3.2 Sensitivity to Compiler Optimization Flags

We now evaluate how sensitive design space exploration is with respect to another characteristic that may affect workload behavior, namely the compiler and its optimizations. Further, we want to understand the relative importance of input versus compiler sensitivity towards workload representativeness.

We therefore consider 250 randomly selected combinations of compiler optimizations flags. As mentioned before, random selection of 250 combinations of compiler optimization flags is found to be a robust way of measuring the performance impact of compiler optimizations [10]. According to the results shown in Figure 4.4, for none of the benchmarks we observe a severe EDP deficiency in the average (and best) case scenario. We observe some EDP deficiency for some of the benchmarks in

the worst case scenario: 16% on average and up to 35%. In other words, the compiler optimizations used to compile the benchmarks may lead to some EDP deficiency if one is unlucky to select a compiler optimization that is non-representative compared to the other optimizations for driving microarchitectural design space exploration. Comparing Figure 4.4 against Figure 4.3, we conclude that microarchitectural design space exploration is much more sensitive to benchmark inputs than to compiler optimizations.

4.3.3 The impact of the microarchitecture on compiler optimization flags

The results in Section 4.3.2 show that microarchitectural design choices are only weakly influenced by the selected set of compiler optimization flags. However, this does not imply that this relationship is symmetric: the results in the previous section *do not* conclude that the selection of compiler optimization flags is only weakly influenced by microarchitectural design choices. In fact, by re-using the same data that we gathered in Section 4.3.2, we are able to show that in contrast to the low sensitivity of microarchitectural design choices on compiler optimization flags, the impact of microarchitectural design choices on the selection of compiler optimization flags is significant. The main intent of this section is to point out this contrast. For strategies on optimizing binaries for a specific microarchitecture we refer to prior research in the end of this section.

To gain initial insight into the potential performance loss when using a set of compiler flags, optimized for a specific microarchitecture, on a different microarchitecture we set up a small-scale experiment, as illustated in Figure 4.5. In this experiment we first use the baseline architecture of Table 4.2 to simulate [3] 250 program binaries for the same application but compiled using different sets of compiler flags and select the program binary with lowest execution time. We refer to this binary as the *baseline-optimal binay*. Second, we simulate the baseline-optimal binary on all 1728 microarchitectures of the design space in Table 4.2 and evaluate the corresponding execution times. Third, we simulate all 250 program binaries on all 1728 microarchitectures and select an optimal binary on each microarchitecture. We refer to a program binary, optimized for a specific microarchitecture, as the *microarchitecture-optimal binary*.

Figures 4.6 and 4.7 reports speedups over `-O3` for the baseline-optimal and microarchitecture-optimal binaries, for the benchmarks lame and sha, respectively. The X-axis sorts the microarchitectures by microarchitecture-optimal speedup. We observe that many microarchitectures benefit from the baseline-optimal binaries (the speedup is larger than one). However,

[3] Simulations performed in this section are done through the model of Chapter 3.

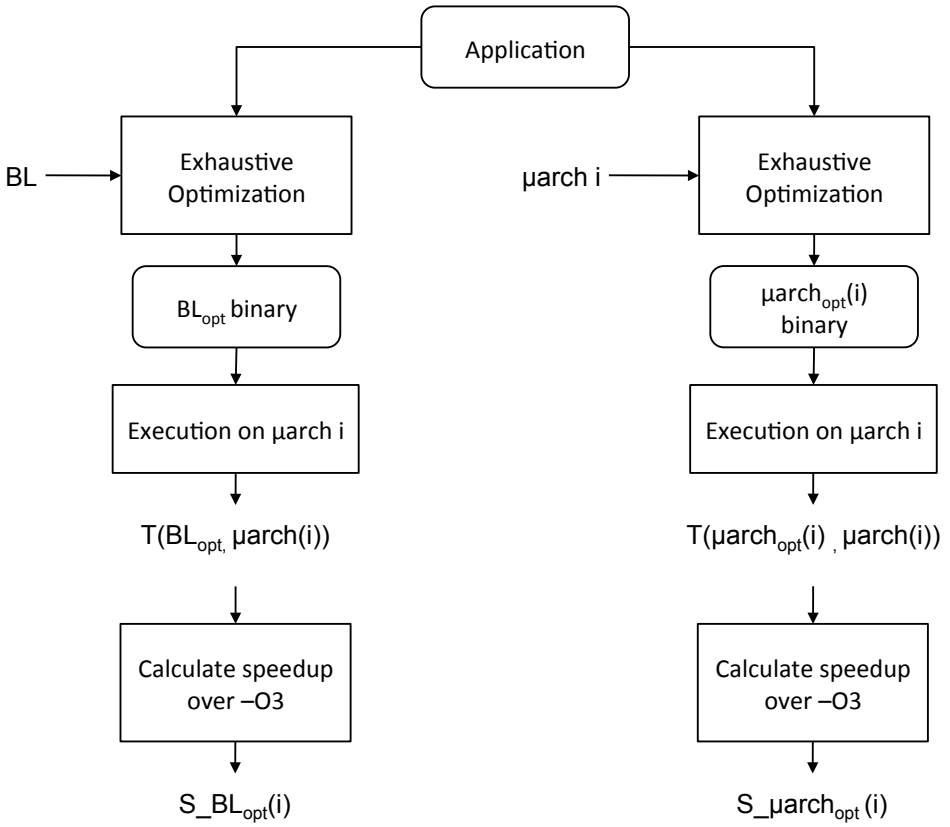


Figure 4.5: Framework to calculate the speedups of the baseline-optimal (BL_{opt}) binary and the microarchitecture-optimal ($\mu arch_{opt}(i)$) binary for microarchitecture i.

as can be seen in the figures, the microarchitecture-optimal binaries have a significant improvement over the baseline-optimal binaries for a number of microarchitectures.

To remove the potential dependence on the baseline architecture we now setup a large-scale experiment where any microarchitecture can be used as a baseline. To ease the quantification we again distinguish three cases and plot their results in Figure 4.8. The best case here is the case where we use microarchitecture-optimal binaries on each microarchitecture. This is the ideal situation and hence it is not possible to achieve a higher speedup by using any of the other 249 binaries. We report this in Figure 4.8 as harmonic mean speedups over all microarchitectures. The worst case refers to selecting a baseline microarchitecture A that results in a baseline-optimal binary with the worst results on the actual microarchitecture B. We repeat this for each microarchitecure B and report the harmonic averages in Figure 4.8. The average case is the average speedup we get by using any microarchitecture A as a baseline to generate a binary for microarchitecture B. We

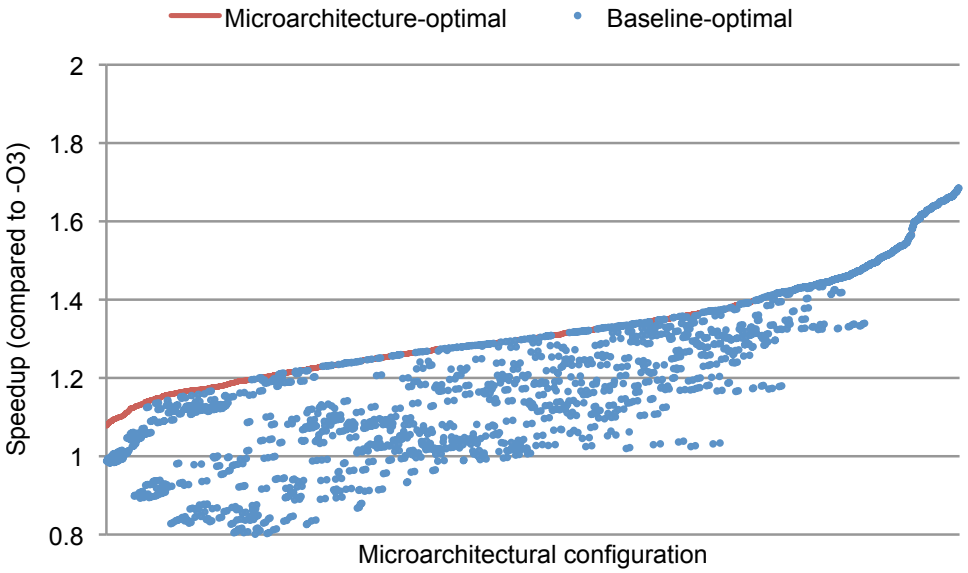


Figure 4.6: Speedups over `-O3` for lame when the binary is optimized for the target architecture (red curve) and when it is optimized by a baseline architecture (blue data points). The experiment is repeated for different microarchitectures on the X-axis, but the baseline is kept the same.

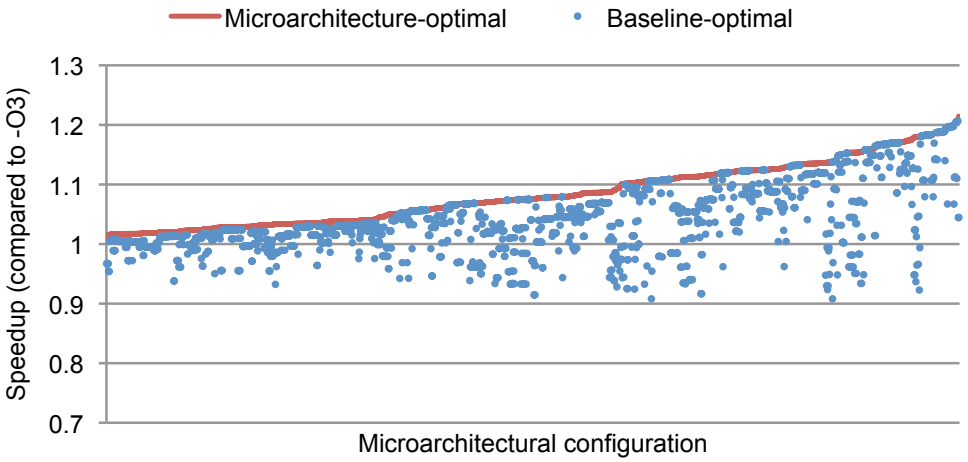


Figure 4.7: Speedups over `-O3` for sha when the binary is optimized for the target architecture (red curve) and when it is optimized by a baseline architecture (blue data points). The experiment is repeated for different microarchitectures on the X-axis, but the baseline is kept the same.

again repeat this for each microarchitecture B and report the harmonic averages in Figure 4.8. We observe that microarchitecture-optimal binaries (i.e., the best case) achieve a speedup of 1.27 on average, while baseline-optimal binaries that result in the worst speedups achieve a speedup of

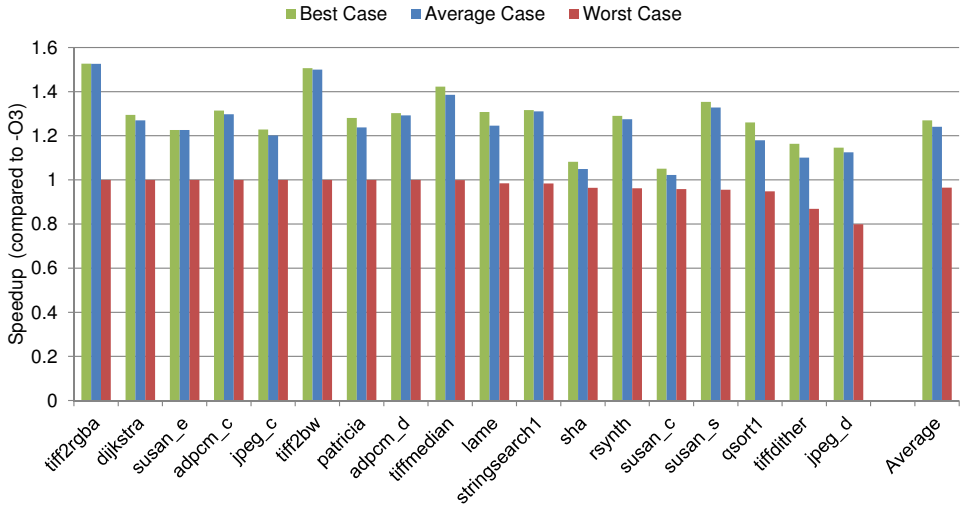


Figure 4.8: Harmonic average of speedups over 1728 microarchitectures, when the baseline to generate the application binary is (1) the microarchitecture resulting in the best speedup (i.e., the binary is generated by the microarchitecture we measure speedup on and hence is the best case.), (2) the microarchitecture resulting in an average speedup (average case) and (3) the microarchitecture resulting in the worst speedup (worst case)

0.97 on average. Hence, on average, we can improve speedup by 30% by generating a microarchitecture-optimal binary, compared to using a binary generated by the baseline resulting in the worst speedup.

These results show that the selection of compiler optimization flags is highly dependent on the underlying microarchitecture. Finding the optimal set of flags has been subject of many research as an exhaustive search of all possible combinations of compiler flags is infeasible. Hoste and Eeckhout [34] use a genetic algorithm to optimize binaries along multiple objectives (e.g., code size and execution time). Other machine learning techniques, such as [15], [51], [53], [60] and [44] aim at predicting the performance of different sets of compiler flags, so that an optimal set can be selected without evaluating execution time. Dubach et al. [17] build a machine learning technique that predicts a microarchitecture-specific optimal set of compiler flags. The model takes a set of performance counters and a microarchitectural description as input.

We conclude that in contrast to the *weak* impact of the selected set of compiler optimization flags on microarchitectural design space exploration, the reciprocal relationship, namely the impact of microarchitectural design choices on the selection of compiler optimization flags is *strong*.

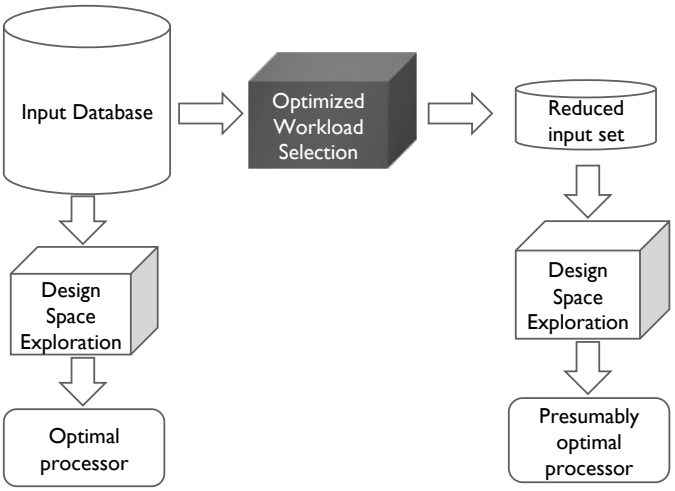


Figure 4.9: Input selection workflow.

4.4 Representative Benchmark Input Selection

We now present and evaluate various methods for selecting representative benchmark inputs. We do this using a common workflow as shown in Figure 4.9. We start off with the input database shown at the top left (i.e., 1,000 inputs per benchmark), from which we select a number of presumably representative inputs, called the *reduced input set*, shown at the top right. We evaluate all possible processor configurations for the selected inputs, and determine the most optimal configuration, which is our *presumably optimal processor*. To evaluate the effectiveness of the input selection method, we compare the presumably optimal processor with the (globally) *optimal processor* obtained by taking into account all inputs during the design space exploration (left hand side of Figure 4.9). We do this by computing normalized EDP for the presumably optimal processor, using Equation 4.4. The closer the normalized EDP to one, the closer the presumably optimal processor is to the optimal processor.

In Subsections 4.4.1 through 4.4.4, we consider four benchmark input selection methods (random selection and three systematic input selection techniques). We end this section with a discussion in subsection 4.4.5 on trading off input selection speed versus efficiency for the various selection techniques.

The inputs that we identified to be representative are presented in Appendix B for the two best techniques.

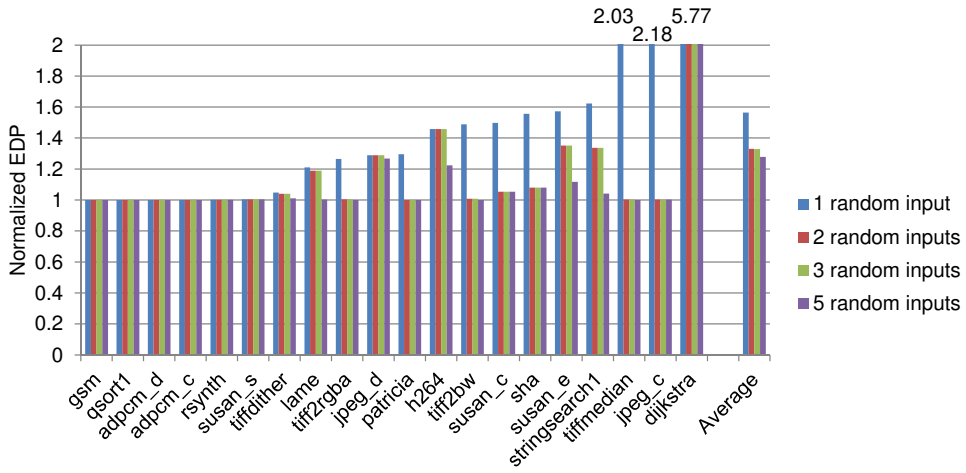


Figure 4.10: Random input selection: worst case normalized EDP as a function of number of randomly selected inputs.

4.4.1 Random Selection

To provide a solid ground of comparison, we first implemented *random selection*, i.e., we select a number of inputs at random out of the pool of available benchmark inputs. More specifically, we evaluate the effect of the number of randomly selected inputs on the presumably optimal design. The expectation is that as we consider more randomly chosen inputs, the more representative this set of inputs becomes and hence the closer the presumably optimal processor will be to the globally optimal processor. This is indeed the case: as we increase the number of selected inputs from 1 to 5, the average worst case EDP decreases from 57% (one input), to 33% (two inputs), and 28% (five inputs), over the globally optimal processor, see Figure 4.10. These results were obtained by taking 1,000 randomly chosen sets of n inputs, with n varying from 1 to 5, and selecting the worst possible set. (Note that selecting one input at random corresponds to the results previously shown in Figure 4.3 under the 'worst case' scenario.)

As we increase the number of randomly selected inputs from one to two, we observe a fairly steep decrease in the average worst case EDP from 57% to 33%, after which adding more inputs delivers diminishing returns. Some benchmarks greatly benefit from selecting a couple of randomly chosen inputs, see for example tiff, patricia and jpeg_c. Unfortunately, there are also quite a few benchmarks for which selecting multiple inputs at random does not solve the problem, see for example lame, jpeg_d, dijkstra and h264. Picking five randomly selected inputs may still lead to a suboptimal design point with an average 28% worst case EDP deficiency compared to the globally optimal processor configuration. Clearly, random input selection is not effective at composing a representative input set, if few inputs

Parameter	*Optimized for input 807*	*Optimized for input 261*
Issue width	2	2
Data cache size	64 KB	16 KB
Data cache associativity	4	2
Instruction cache size	16 KB	32 KB
Instruction cache associativity	4	2
Cache block size	32 byte	64 byte
Branch predictor	3.5 KB hybrid	3.5 KB hybrid
Pipeline Depth	9	9

Table 4.4: Resulting microarchitectural optimizations for jpeg_d, depending on the used input during design space exploration.

are to be selected. Picking many more inputs is likely to solve this issue, however, it comes at the cost of requiring substantially longer simulation times during design space exploration. Instead, we devise other input selection methods that are more effective at selecting a few representative inputs than random selection.

4.4.2 Microarchitecture-Independent Selection

Our second input selection mechanism, *microarchitecture-independent selection*, selects a pair of representative inputs after characterizing the inputs in the input database through microarchitecture-independent characterization. Characterizing inputs incurs some overhead compared to random selection (which incurs no overhead at all), however, the cost is a one-time cost only, i.e., each input needs to be characterized only once. Fortunately, this characterization step is much faster than detailed cycle-accurate simulation. The advantage of using a microarchitecture-independent characterization is that it characterizes the inputs in such a way that the characterization can be leveraged across microarchitectures and processor configurations.

Before going into the details of microarchitecture-independent characterization we first analyze the program-behavior of jpeg_d through valgrind [49] for two different inputs as an example.

Valgrind [49] is a tool that profiles an application to build a call graph that stores information such as dynamic instruction counts. When visualizing the call graph only the subroutines with the highest fraction of instructions are shown to ease the analysis.

Figure 4.11 and 4.12 show input 807 and input 261, respectively. Using input 807 to optimize the microarchitecture for jpeg_d in terms of EDP results in a microarchitecture that has a normalized EDP of 1 when evaluated

Figure 4.11: Input 807 for jpeg_d

Figure 4.12: Input 261 for jpeg_d

for all other inputs according to Formula 4.4. This means that input 807 results in a microarchitecture that is optimal over all inputs. If we use input 261 to optimize the microarchitecture however, and evaluate the resulting data set-optimal microarchitecture for all other inputs we find a normalized EDP of 1.27. This means that input 261 results in a microarchitecture that has an EDP 27% higher than the optimal microarchitecture. The two different microarchitectures can be seen in Table 4.4.

From Figures 4.11 and 4.12 it is not apparent why one of the images would result in better overall design decisions than the other. However,

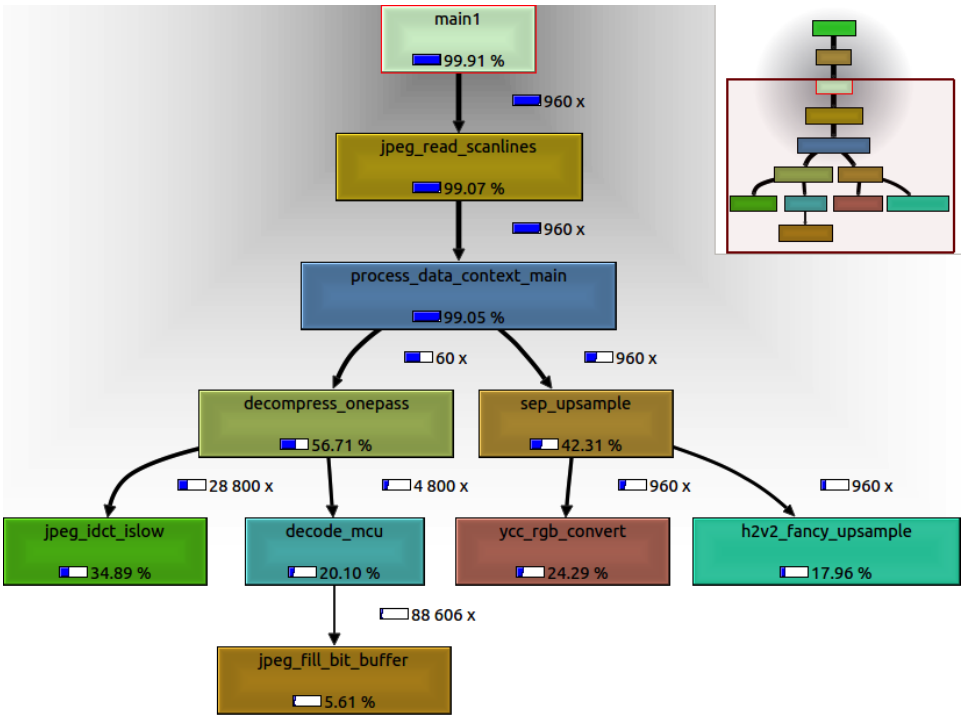


Figure 4.13: Call graph of the valgrind analysis for input 807

when we look at the valgrind call graphs in Figure 4.13 and 4.14 for input 807 and 261, respectively, we observe very different program-behavior.

From Figure 4.14 we can see that about 40% of all instructions are spent in jpeg_start_decompress. Despite what the name suggests, this is not initialization code. In fact 34 million instructions are spent in jpeg_start_decompress. The reason is that jpeg_start_decompress comprises a number of decoding steps that are specific for images encoded with the multi-pass coding scheme. In contrast, input 807 is encoded with the single-pass encoding scheme and hence skips most of the subroutines inside jpeg_start_decompress so that a total of less than 20 thousand instructions are spent inside the routine. Comparing Figures 4.13 and 4.14 further, we observe that process_data_context_main has different decompress variants for single-pass and multi-pass encoded images.

Clearly, it requires a lot of domain-specific expertise or deep analysis to understand the difference in program-behavior for different inputs. In order to make it transparant to the microarchitect how different inputs behave, we need to be able to automate the detection of inputs that have a different behavior.

To characterize the behavior we rely on the concept of a basic block vector (BBV), as previously introduced by [56]. A basic block is an atomic piece

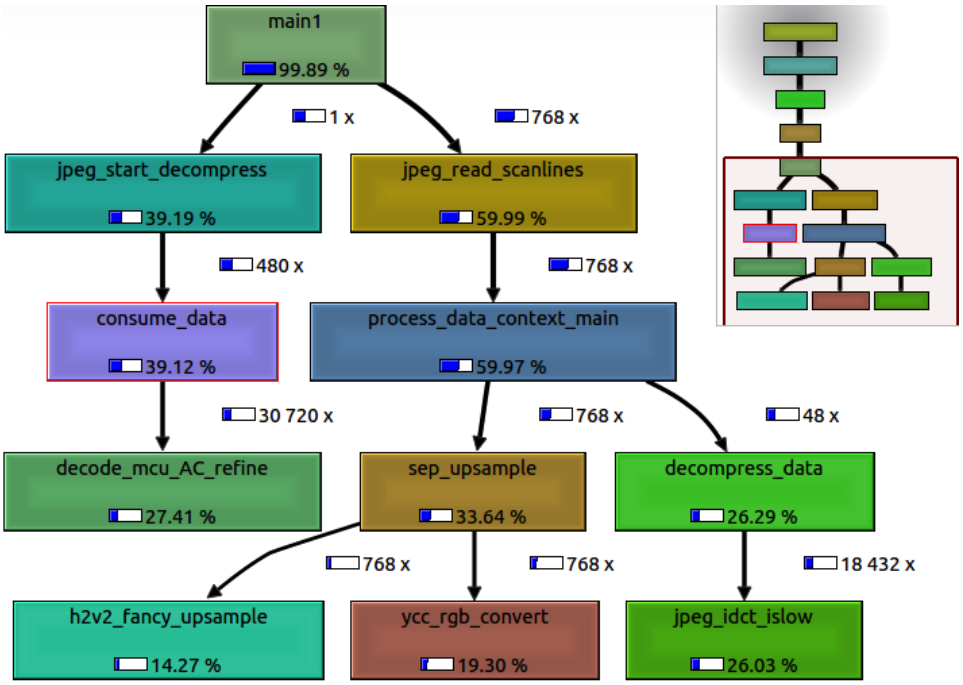


Figure 4.14: Call graph of the valgrind analysis for input 261

of code with a single entry and exit point. A basic block vector is a vector that keeps track of how often each basic block is being executed dynamically. The dimensionality of a BBV equals the number of basic blocks in a program and each index in the BBV counts how many times the respective basic block is executed for a given input. In other words, a BBV represents which blocks of code are being executed, and how frequently. We normalize a BBV so that the absolute value of the vector equals one. We compute a BBV for each input, and use it to characterize the dynamic behavior of the program and the given input in a microarchitecture-independent way. Previous work has demonstrated the good correlation between BBVs and dynamic execution behavior [43]. We build on this prior work, and use BBVs to understand (dis)similarities among inputs.

The microarchitecture-independent input selection method we propose is comprised of three steps to create a reduced input set of two inputs, see also Figure 4.15. We first characterize each input through a BBV, i.e., we execute the benchmark and its input and count how often each basic block is being executed — this can be done through functional simulation (using an architectural simulator such as gem5, as we did), or through a dynamic binary instrumentation tool such as Pin [47]. We subsequently normalize the BBVs. Next we calculate the Manhattan distance between the BBVs of each possible pair of inputs. The Manhattan distance between the BBVs of

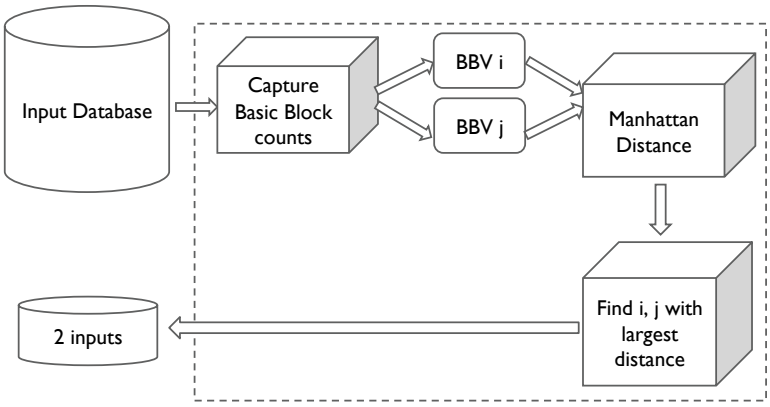


Figure 4.15: Microarchitecture-independent input selection using Basic Block Vectors (BBVs).

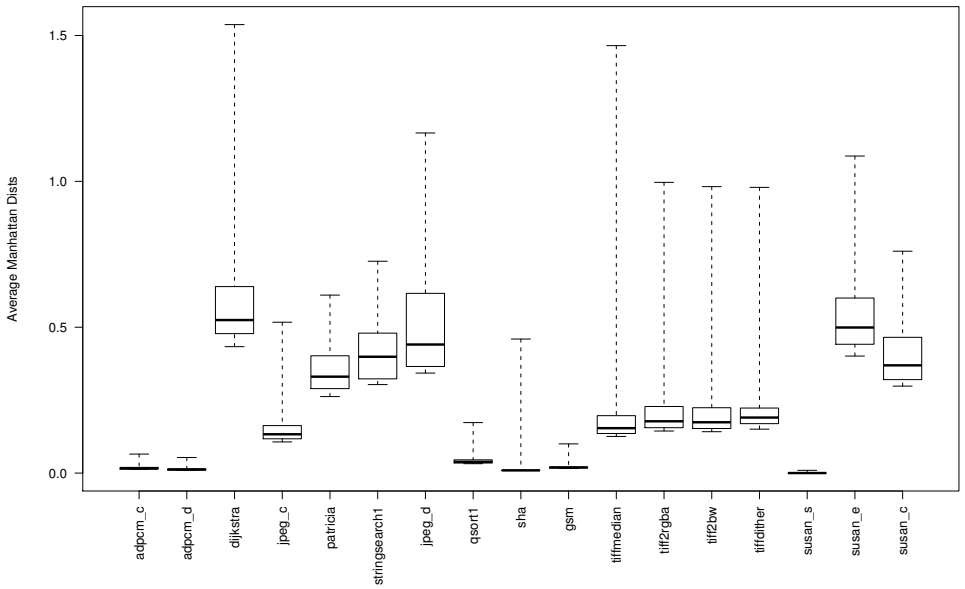


Figure 4.16: Boxplot distribution of average Manhattan distances between each input and all other inputs. The high distances indicate that there is many different dynamic behavior over different inputs: e.g., a Manhattan distance of 0.5 indicates that 25% of the executed code resides in different basic blocks.

inputs i and j is calculated as follows:

$$MD(BBV_i, BBV_j) = \sum_{k=1}^{N} |BBV_i(k) - BBV_j(k)|, \tag{4.5}$$

with N the dimensionality of the BBV, or the total number of static basic blocks in the program. Figure 4.16 illustrates the distribution of average Manhattan distances between all inputs. In the third and last step, we de-

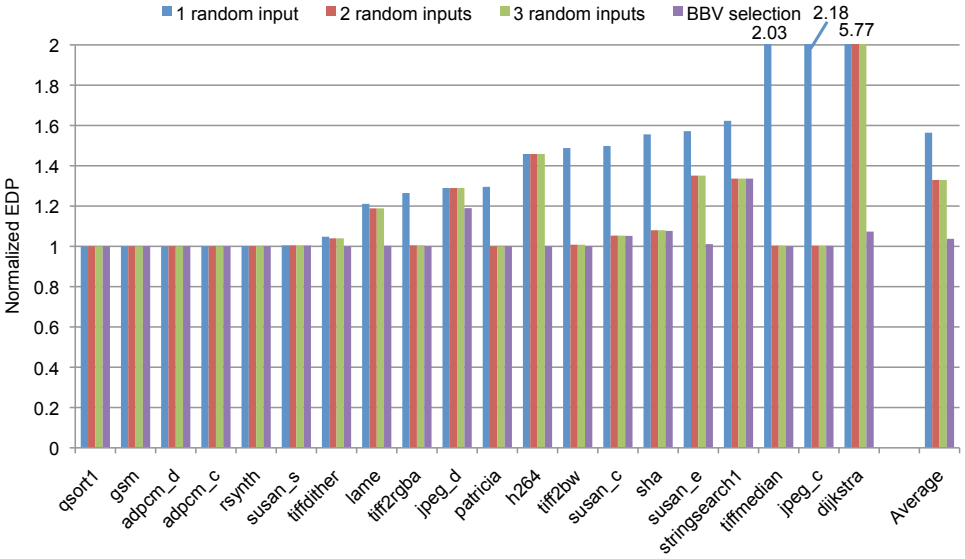


Figure 4.17: Normalized EDP for microarchitecture-independent input selection using BBVs versus random selection.

termine the pair of inputs (i, j) with the biggest Manhattan distance (MD):

$$(i, j) = \arg\max_{i,j}(MD(BBV_i, BBV_j)). \tag{4.6}$$

Intuitively, this pair denotes inputs with the most divergent code execution behavior, which is likely to lead to diverse run-time behavior and is therefore likely to be more representative than randomly selected inputs.

Figure 4.17 compares the microarchitecture-independent selection using BBVs, as just described, against random selection. On average, this method results in a presumably optimal design point that is no more than 3.7% off compared to the globally optimal design point, which is a substantial improvement over random selection (e.g., 33% off on average for two randomly selected inputs). The maximum EDP deficiency for the BBV selection method is observed for stringsearch (33.6%); the reason why we are seeing relatively high deficiencies using the BBV method (see jpeg_d and stringsearch) might be that a BBV only tracks the code being executed and not how it interacts with the microarchitecture (e.g., different memory access behavior may be observed even though the same code is being executed). For all applications we are better off (or equally good) using BBV selection compared to randomly selecting two inputs. In fact, we even do better (or equally good) compared to three randomly selected inputs. This makes the microarchitecture-independent selection method using BBVs a reliable method for selecting representative inputs for design space exploration.

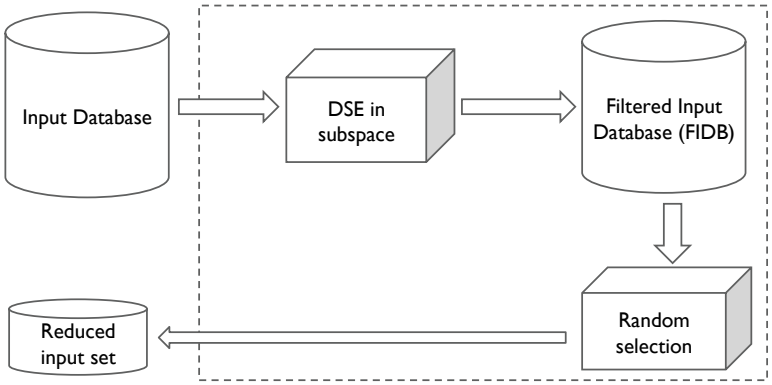


Figure 4.18: One-level filtered input selection uses design space exploration in a limited microarchitectural subspace to filter out non-representative inputs.

4.4.3 Filtered Selection

The third input selection method combines design space exploration and input filtering before randomly selecting a number of inputs. The basic idea is that if we can filter out non-representative inputs, we can greatly reduce the worst case scenario when randomly picking inputs. We have developed two filtered selection methods, which we describe next: one-level and two-level filtered selection. Both techniques use all inputs to perform a design space exploration in a limited subspace. Inputs that lead to far from optimal design points in the limited subspace are filtered out as it is likely that they are unrepresentative of the larger design space. The resulting filtered input database (FIDB) is then used to randomly select inputs.

One-level filtered selection

The one-level filtered selection method performs design space exploration with all the inputs on a small subspace of the larger microarchitectural design space, as shown in Figure 4.18. The design space exploration in this subspace involves estimating performance and power/energy for all design points in the subspace and for all inputs in the input database. We consider a subspace of 10 design points; these design points include a baseline configuration (conf0 from Table 4.1) along with 9 other design points derived from this baseline by randomly changing a number of the microarchitecture parameters shown in Table 4.2, one at a time. Once the exhaustive evaluation is done in this subspace, we filter out the inputs that result in poor average normalized EDP, relative to the optimum design point in the subspace. This results in a filtered input database (FIDB). The goal is that the FIDB no longer contains inputs that may lead to poor design points in the larger design space. From the FIDB, we then select a number of in-

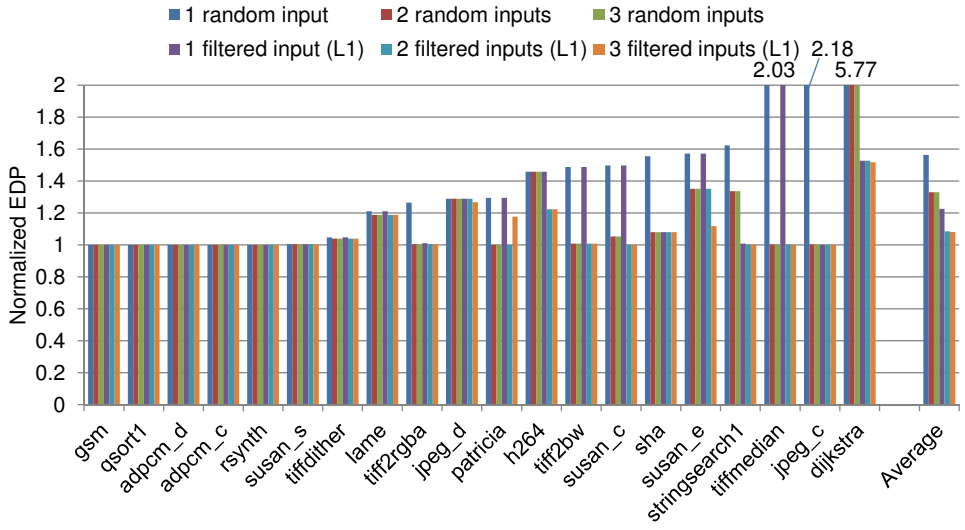


Figure 4.19: Normalized EDP through one-level filtered input selection.

puts at random.

Figure 4.19 reports worst case normalized EDP for the one-level filtered selection method when randomly picking 1, 2 and 3 inputs from the FIDB. One-level filtered input selection leads to average worst case EDP deficiencies of 22.6%, 8.6% and 8.1%, respectively, which is a substantial improvement over random selection. This confirms that performing a preliminary design space exploration in a small subspace of the larger design space filters out a number of non-representative inputs. For most benchmarks between 0 and 200 inputs are filtered out, i.e., the FIDB contains between 800 and 1000 inputs. However, for two benchmarks (dijkstra and jpeg_c), the FIDB contains slightly more than half of the original input database. The cost of one-level filtered selection is a one-time cost involving detailed simulation of all design points in the subspace for all inputs. In our setup with 1,000 inputs and 10 design points in the subspace, this results in 10,000 evaluations per benchmark.

Two-level filtered selection

To further improve the effectiveness of the filtered input selection method, we now consider two levels of filtering, see also Figure 4.20. As with the one-level filtered selection method, we first rely on a design space exploration in a limited subspace. However, instead of selecting a single subspace, we select k subspaces. We randomly divide the original input database in $(k-1)$ sets of inputs and we perform an exhaustive evaluation in $(k-1)$ subspaces. This leads to $(k-1)$ level-one (L1) filtered input

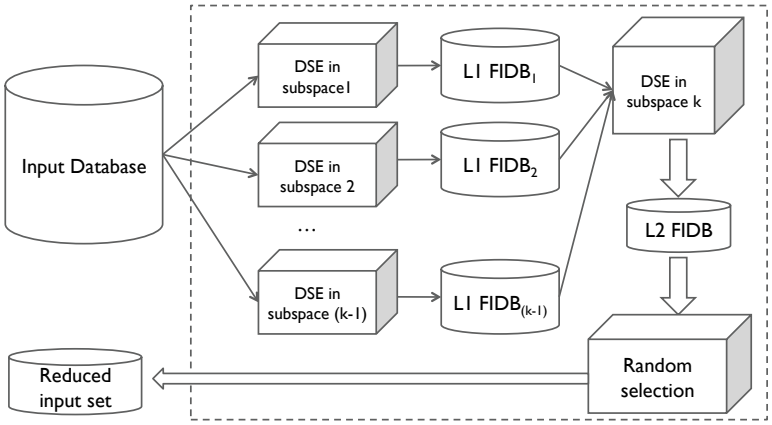


Figure 4.20: Two-level filtered input selection uses two levels of design space exploration in limited subspaces to filter out non-representative inputs.

databases (FIDBs). Next, we randomly select a number of inputs from each of the FIBDs, which we use to exhaustively evaluate the k-th subspace. This leads to a second level of filtering, yielding the level-two (L2) FIDB. We then select a number of inputs at random from the L2 FIDB.

In our implementation we aimed at having the same number of simulations as for one-level filtered input selection, namely 10,000 evaluations per benchmark. We therefore set k to 10 and simulate 100 inputs per subspace. Each subspace is randomly selected and consists of 10 design points. For most benchmarks the L1 FIDBs contain at least 56 inputs of the original 100 inputs. One benchmark, sha, has an L1 FIDB of 17 inputs only. Some benchmarks have a couple of L1 FIDBs with around 60 inputs only while the other FIDBs contain around 100 inputs. This implies that the selected subspace influences the selection of inputs and having multiple subspace could potentially lead to better results than using a single subspace. To construct the L2 FIDB we select 100 random inputs across all the L1 FIDBs. The size of the L2 FIDB ranges between 56 and its maximum possible value of 100.

Figure 4.21 reports that picking one random input from the L2 FIDB leads to an average worst case EDP deficiency of 12.9% over the globally optimal design point, which is a substantial improvement over random selection and one-level filtered selection. This implies that two-level filtered selection effectively does a better job filtering out non-representative inputs. Randomly picking two and three inputs from the L2 FIBD leads to an average worst case EDP deficiency of 8.7% and 6.7%, respectively.

It is worth noting that the input filtering approach, while effective on average, is not perfect. We observe a clear benefit from filtering for all benchmarks, except for patricia. In particular, for patricia, when consider-

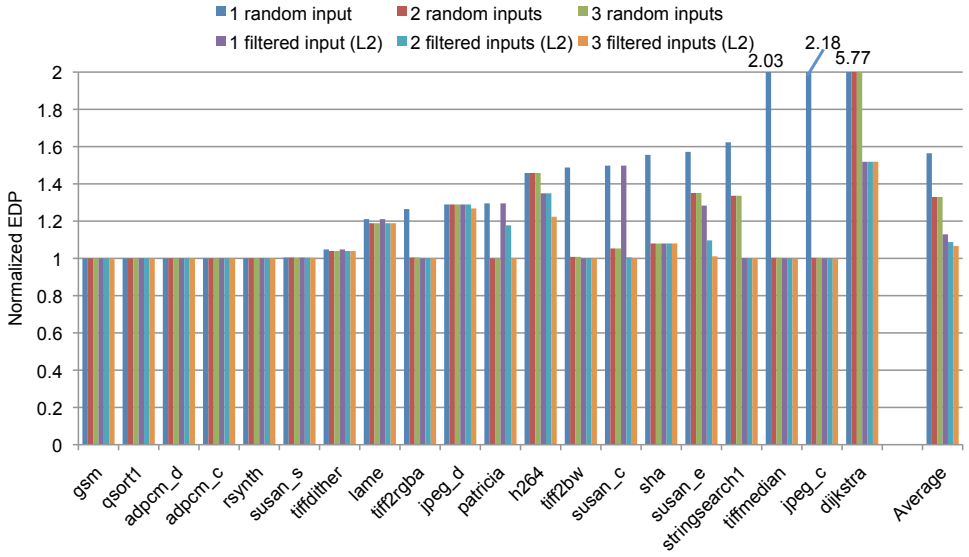


Figure 4.21: Normalized EDP through two-level filtered input selection.

ing two inputs, random selection outperforms two-level filtering; similarly, for three random inputs, random selection outperforms one-level filtering; likewise, for two inputs, one-level outperforms two-level filtering. This counterintuitive result suggests that we may be discarding representative inputs, which may be due to the small subspaces used to filter out inputs, i.e., the subspaces may not be representative for the larger space. A systematic way of determining subspaces may solve this issue, however, we leave this for future work as the random subspace selection process seems to work well for most benchmarks in our study.

4.4.4 CPI-Sampled Selection

The filtered input selection method, while effective, has two major shortcomings. First, the cost for building the FIDB is quite high, i.e., it requires detailed evaluation of all inputs on a number of design points. Second, there is a random component to the selection of inputs from the FIDB. We now propose an input selection method, called *CPI-sampled selection*, that overcomes these issues. It involves a single evaluation of a particular design point only, and selects representative inputs in a systematic way, not through random selection.

The CPI-sampled selection method was inspired by filtered selection, but instead of evaluating subspaces, we consider a single baseline design point only, on which we evaluate all inputs. As we have a single design point only, we have no means of filtering out non-representative inputs based on their performance with respect to ranking design points in the

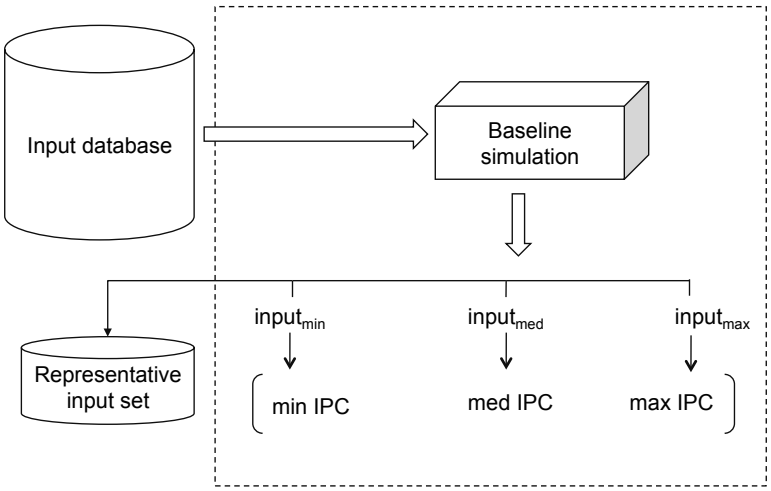


Figure 4.22: CPI-sampled selection.

subspaces. We therefore resort to relative performance of the input compared to the other inputs in the input database, i.e., we pick inputs based on the achieved CPI (cycles per instruction) on the baseline architecture. When selecting one input, we pick the input that has the median CPI across all inputs; when selecting two representative inputs we pick the input with median CPI and another input that has a value as close as possible to the median CPI[4]; when selecting three inputs, we pick the inputs with the minimum, median and maximum CPI value. See Figure 4.22 for a schematic overview of the CPI-sampled selection method.

The CPI-sampled method is effective at identifying representative inputs, as shown in Figure 4.23 for one input (with median CPI), two inputs (with median CPI and nearly-median CPI) and three inputs (minimum, median and maximum CPI). One input leads to an average worst case EDP deficiency of 2% compared to the globally optimal design point (with one outlier of 22% for h264). Two inputs brings the worst case EDP deficiency further down to 1% on average (and at most 7% for sha). Three inputs leads to no EDP deficiency, i.e., the three representative inputs lead to the same optimum design point as using all inputs. The intuition is that the three most extreme inputs, the ones with the minimum, median and maximum performance on a baseline design point, are representative of the entire input database — the other inputs can be considered 'interpolations' with respect to the extreme inputs — which leads to effective design space exploration.

Figure 4.24 shows CPI values for the baseline configuration of a number

[4]We found selecting two inputs close to the median CPI to outperform selecting the inputs with the minimum and maximum CPI.

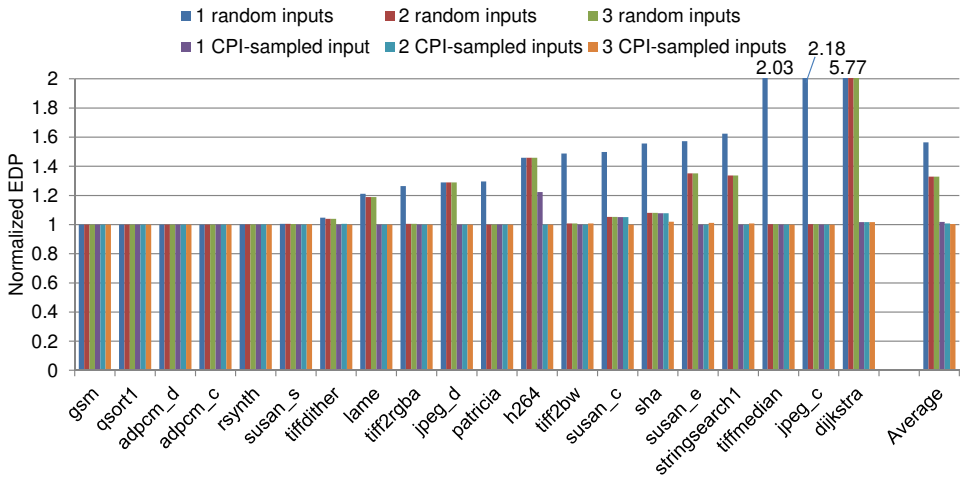


Figure 4.23: Normalized EDP through CPI-sampled selection.

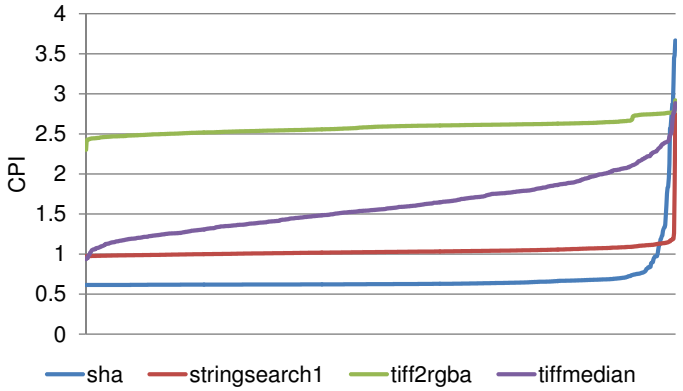


Figure 4.24: CPI across data sets for a number of benchmarks.

of benchmarks across all of their respective inputs, and provides intuition why for some of the benchmarks it is better to pick inputs with median and near-median CPI rather than inputs with minimum and maximum CPI. Benchmarks such as tiffmedian and tiff2rgba (and some others which are not displayed due to space reasons) that have an S-shaped curve for their CPI values, tend to have good results when selecting inputs with minimum and maximum CPI. Benchmarks sha and stringsearch (and others not displayed due to space reasons) on the other hand are L-shaped and selecting inputs with minimum and maximum CPI tend to perform poorly on these benchmarks. The reason is that there are very few inputs with a relatively high CPI. Selecting these inputs puts too much weight on these inputs that occur infrequently and are non-representative for the rest of the inputs. Adding the inputs with median CPI addresses this issue.

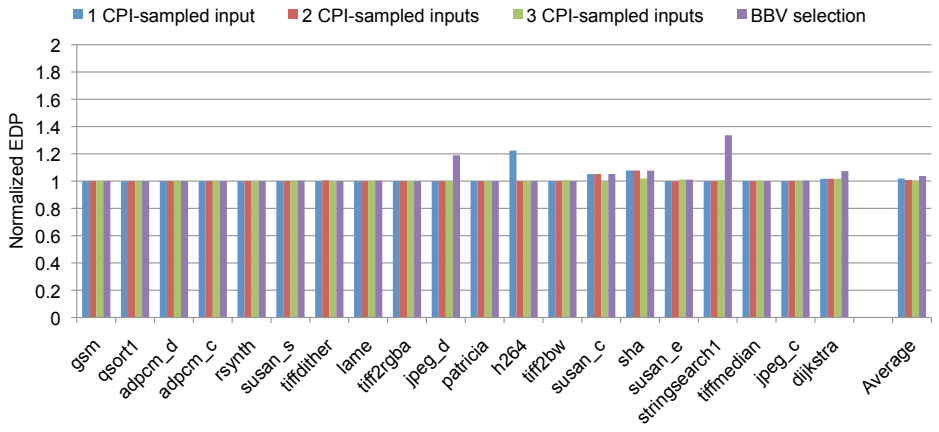


Figure 4.25: Overview of the best performing two techniques: BBV selection and CPI-sampling

4.4.5 Overview and Discussion

Table 4.5 summarizes and compares the proposed input selection methods, while Figure 4.25 provides an overview of the two best performing techniques. Note that in Table 4.5 we also included the complexity as a function number of inputs in the database N, the number of microarchitectural configurations M, the simulation time overhead for a single simulation s, and the instrumentation overhead p for a single instrumentation run. Typically p is much smaller than s.

While random selection incurs no cost for selecting inputs, it may lead to severe EDP deficiencies. Even selecting three inputs at random may result in an average worst case EDP deficiency of 33% compared to using all inputs. The filtering method is more effective at selecting representative inputs, with an average worst case EDP deficiency of 6.7% for the two-level approach and three randomly selected inputs. However, filtering comes at the cost of requiring $10 \times N$ detailed evaluations (detailed simulations), with N the number of inputs in the input database. The microarchitecture-independent selection method using BBVs incurs a one-time cost of collecting BBVs for all inputs. Fortunately, collecting BBVs is essentially a profiling step which is much faster than detailed simulation. The BBV selection method is more effective than filtering. It achieves an average worst case EDP deficiency of 3.7% while using only two inputs during design space exploration; we observed an EDP deficiency of 33.6% for one benchmark though. CPI-sampled selection is the most effective approach: selecting the inputs with the minimum, median and maximum CPI for a baseline configuration leads to identifying the globally optimal design point for all benchmarks. This comes at the cost of requiring N detailed evaluations, which

Selection method	*One-time cost*	*#Inputs*	*Complexity*	*Avg worst case*	*Max worst case*
Random selection	None	1 - 2 - 3	$O(s \times M)$	57% - 33% - 33%	477% - 477% - 477%
One-level filtering	$10 \times N$ detailed evaluations	1 - 2 - 3	$O(s \times N)+$ $O(s \times M)$	23% - 9% - 8%	103% - 53% - 52%
Two-level filtering	$10 \times N$ detailed evaluations	1 - 2 - 3	$O(s \times N)+$ $O(s \times M)$	13% - 9% - 7%	52% - 52% - 52%
BBV selection	Instrumentation of N inputs	2	$O(p \times N)+$ $O(s \times M)$	4%	34%
CPI-sampled selection	N detailed evaluations	1 - 2 - 3	$O(s \times N)+$ $O(s \times M)$	2% - 1% - 0%	22% - 8% - 2%
Exhaustive selection	None	N	$O(s \times N \times M)$	0%	0%

Table 4.5: Summary of input selection methods. (N is the number of inputs in the database, M the number of microarchitectures, s the simulation time overhead and the p instrumentation overhead.)

is more time-consuming than BBV profiling. Note that we also added the characteristics of exhaustive search. Exhaustive search comes down to exploring the whole design space with all possible inputs. This leads to a perfect design at the highest exploration cost, which is infeasible.

These results lead to two important conclusions or insights. First, it is important to select benchmark inputs in a systematic way. Randomly selected inputs, even from a subset of inputs as with the filtering approach, may lead to non-representative behavior, which eventually leads to non-optimal design points during design space exploration. Selecting inputs based on their behavior, either through microarchitecture-independent characterization (e.g., BBV selection) or through detailed measurements on a particular system (e.g., CPI-sampled selection), improves the representativeness of the workload, ultimately leading to a better final design. Second, the two most accurate input selection methods involve a trade-off. CPI-sampled selection is (slightly) more effective compared to BBV selection at identifying the optimum design point, however, this comes at the cost of incurring more overhead as it requires detailed measurements for all inputs on a given baseline design point.

Although the presented techniques have only been evaluated for superscalar in-order processors, we believe that they also apply (and might be even more important) for more complex microarchitectures (e.g., out-of-order processors). Performance might be more sensitive to inherent workload behavior and input selection as the architecture is more complex, hence a broader set of inputs might be needed compared to in-order processors. If needed, some of the proposed techniques might need minor adjustments to select a slightly broader range of representative inputs. For example, the CPI-sampled selection method can be extended by adding inputs between the minimum and median, and between the median and maximum, to have a more fine grained selection of inputs. When looking at multi-core processors, special attention should be given regarding how to best select inputs to form representative multi-program and multi-threaded workloads. Prior work has selected multi-program workloads randomly [64], or based on microarchitecture-independent characterization of benchmarks [63]. Extending and/or combining these techniques with techniques for selecting representative inputs is left for future work.

4.5 Summary

This chapter provides a comprehensive study of the impact of two implicit workload parameters on microarchitecture design space exploration: benchmark inputs and compiler optimization flags. Whereas we observe only a limited impact from the used compiler optimization flags, bench-

mark inputs can impact design space exploration significantly. To address this issue, we introduce three novel benchmark input selection techniques.

Our setup to study input sensitivity for design space exploration involves 1,000 inputs for 20 embedded benchmarks and a design space consisting of around 1,700 design points. We use EDP as our optimization criterion, and we focus on the average EDP deficiency across all inputs, when the worst possible input(s) are used during design space exploration. This is a case that should be avoided at all cost since it is unknown a priori whether the selected inputs are representative or not. We find that random selection may lead to an average worst case EDP deficiency of 57% and 33% when 1 and 3 inputs are randomly selected, respectively. To tackle this problem, we propose filtered input selection, BBV selection and CPI-sampled selection. Filtered selection, when two levels are deployed and 3 random inputs are selected, reduces the worst-case EDP deficiency to 6.7% on average. BBV selection and CPI-sampled selection are more effective than filtered selection, and in addition incur a lower overall cost by characterizing the behavior of the inputs. BBV selection, which characterizes inputs microarchitecture-independent using code execution frequencies, achieves an average worst case EDP deficiency of 3.7% with two inputs selected. CPI-sampled selection, which characterizes inputs microarchitecture-dependent using simulation statistics, finds the globally optimal design point in our design space with three inputs. Hence, CPI-sampled selection results in (slightly) more representative benchmark inputs than BBV selection, at the cost of higher overhead (cycle-level simulation versus profiling).

Overall, we hope this work increases awareness to benchmark inputs and its importance to design space exploration. Given how data driven architecture research and development is nowadays, it is of paramount importance to pay close attention to selecting representative benchmark inputs. We believe this is going to be increasingly important as we resort to application-specific and domain-specific specialization of processor hardware to sustain the performance and energy-efficiency growth curve in the absence of Dennard scaling.

Chapter 5

Conclusion

The important thing is not to stop questioning.
Albert Einstein

In this thesis we covered two key contributions that help computer architects speed up the design space exploration and build microprocessors that are robust across workloads for important design goals such as performance and energy efficiency.

First, we have built a fast offline analytical model to focus design space exploration into areas for detailed simulation and to gain insight into application-microarchitecture interactions. We showed that the model is accurate within 2.8% on average when compared to detailed cycle-level simulation and within 10% when compared against hardware. In addition, because the analytical model builds on the internal mechanics of the microarchitecture, it can break down the total execution time into smaller terms that provide a better level of detail into how the application interacts with the microarchitecture.

Second, we developed several techniques to select a small number of benchmark inputs that are representative for the application's dynamic behavior. We showed that a benchmark application's input data sets can have a significant impact on design decisions in terms of energy-delay-product (EDP): without a systematic selection technique the EDP can be 57% higher on average than the design with lowest EDP. With the techniques introduced in this thesis we are able to find a design with an EDP that is less than 3.7% higher than the design with lowest EDP, without a significant increase in simulation time.

5.1 Summary

5.1.1 Analytical performance modeling

Designing a microprocessor is a challenging task as microprocessors consist of many different components which all need to be adjusted to meet the design goals for a large range of workloads. This makes the processor extremely parameterizable and hence, computer architects use cycle-level microarchitectural simulators to get performance estimates for a given set of configurations. Unfortunately, simulating applications on microarchitectural simulators is up to 4 orders of magnitude slower than executing them on real hardware, which limits the number of microarchitectural designs that can be evaluated in a given amount of time.

Therefore, we presented an analytical performance model in Chapter 3, which evaluates the performance of several microarchitectural configurations in a couple of seconds at a small accuracy loss. This allows us to use the model to explore a large design space and identify a smaller set of interesting designs that is reasonable to be further evaluated with detailed simulation.

More specifically, we built a mechanistic performance model for superscalar in-order processors. Previous work on mechanistic modeling was limited to out-of-order processors. The challenge in modeling in-order processors is the fact that their throughput of instructions is frequently interrupted by inter-instruction dependencies and functional unit contention. Out-of-order processors suffer less from resolving inter-instruction dependencies and waiting on functional units to become available as they are built to hide these latencies. The model consists of an analytical formula which is built on the internal mechanics of the microprocessor. The input to the model is a profile consisting of cache miss rates, branch misprediction rates and a matrix containing fine-grained information on dependence distances and functional unit access patterns; and the machine parameters of interest, such as superscalar width, processor depth, etc. Capturing the profile needs to be done only once for each application to get performance estimates of a wide range of microprocessors and is more than one order of magnitude faster than a single cycle-level simulation. Our results show an absolute error of 2.8% on average and a maximum error of 13% when compared to cycle-level simulation. When compared against hardware, by using a BeagleBone Black board with an ARM Cortex-A8, we show that the model has an absolute error of 10% on average, without making fundamental modeling changes.

We illustrate the model's usefulness by using it as a design space exploration tool but also as a tool to get insight into the application-microarchitecture interaction. Using it as a design space exploration tool

we are able to find a minimal set of functional units for a given performance target. Further, the model allows us to find a microprocessor configuration with an EDP within 1% of the configuration with lowest EDP, found with cycle-level simulation. The model is also used to create cycle stacks, which provides a breakdown of the total number of executed cycles into smaller terms that provide a better level of detail into how the application interacts with the microarchitecture. This breakdown allows us to identify main sources of performance penalty for several benchmarks and understand how compiler optimizations affect the application-microarchitecture interaction.

5.1.2 Selecting representative benchmark inputs

It is a well known fact that the selected set of workloads is key in performing microarchitectural design space exploration. Therefore, much prior work has focused on selecting representative benchmark applications [19, 40, 52, 68, 37, 69]. Benchmark applications are however subject to two implicit parameters: the input to the application and the set of compiler flags that is used to optimize the application's binary. Common practice is to use one or a couple of inputs and application binaries, and assume that they capture enough dynamic behavior to be representative for the entire application. These assumptions are never proven to be true, and the need to study them is increasing when moving toward application-specific processors.

Chapter 4 studies the impact of both these factors, using input databases of 1,000 input data sets, and 250 randomly selected compiler optimization flags. We have shown that using unrepresentative benchmark inputs can lead to an EDP deficiency of 57%, while poorly selected compiler flags can result in an EDP deficiency of 16%. The high potential impact of benchmark inputs on design space exploration suggests the need for systematic benchmark input selection techniques. Therefore we have developed three different systematic input selection techniques to help computer architects build microprocessors that meet the design goals for a wide range of inputs.

The first technique, BBV selection, selects two inputs with the most divergent behavior by comparing the basic block vectors (BBVs) for different inputs. Because calculating the BBVs can be done quickly through functional simulation or through binary instrumentation, it introduces a very low one-time cost. Further, because BBVs are inherent to the application's binary, this is a microarchitecture-independent selection technique. Our experimental results show that we can reduce the potential EDP deficiency to 4%, by using the two inputs selected with BBV selection.

The second technique, filtered selection, performs design space explo-

ration on a limited set of designs in the larger design space. It uses these results to filter out non-representative inputs, inputs that would have resulted in undesirable design decisions in the limited design space. From the remaining inputs a number of inputs is then randomly selected to be used through further design space exploration. The required one-time cost is $N \times S$ simulations, with N being the number of available/generated inputs, and S the number of designs in the limited design space. We find that for one level of filtering we can reduce the EDP deficiency to 9% for two inputs and to 8% for three inputs. For two levels of filtering we can reduce the EDP deficiency to 9% and 7% for two and three inputs, respectively.

The third technique, CPI-sampled selection, uses performance metrics from simulation on a baseline architecture (here CPI values), to select inputs. Inputs are ranked based on their CPI-values and the desired number of inputs is selected based on their ranking. The technique requires N simulations on a baseline configuration, with N the number of inputs available. Because the technique requires simulation on a baseline architecture, this is a micro-architectural dependent technique. When using two inputs, we find that CPI-sampled selection reduces the EDP deficiency to 1%. For three inputs, CPI-sampled selection finds the minimal EDP for nearly all benchmarks in our setup.

5.2 Future work

5.2.1 Mechanistic performance modeling of superscalar in-order processors

The proposed mechanistic model focuses on single-core processors only. It would be interesting to extend this work towards multi-core processors with superscalar in-order cores. The challenge with extending towards multi-core processors is accounting for extra cache misses and longer memory access times because of resource sharing. To be able to use the model in a multi-core environment, parameters corresponding to cache misses and memory access times need to be estimated before evaluating Formula 3.2. Existing work such as [9] and [64] propose techniques to calculate these estimates and are orthogonal to our work

Another interesting extension would be towards multi-threaded processors. Multi-threaded processors are built to hide stall latencies by executing multiple applications on the same processor. This means that next to resource contention, we need to model the overlapping of stalls from the different application threads. Chen and Aamodt [9] model multi-threaded processors for *single-issue* in-order cores. They implement a Markov Chain that transitions between states, where each state indicates how many

threads are stalled. By calculating the possibility that all threads are stalled at the same time, we can estimate the overall throughput. The inputs to the model are IPC results of the individual applications and stall-events. Eyerman and Eeckhout [22] use similar inputs (i.e., single-thread statistics) to calculate overall throughput for multi-threaded out-of-order processors.

The challenge when considering *superscalar* (i.e., multi-issue) multi-threaded in-order processors, is that instructions from different applications could appear simultaneously in the same pipeline stage and hence our detailed instruction-mix profile (i.e., the H-matrix) is dependent on the runtime context. A possible approach to solve this is, instead of computing a single profile for each application, to compute multiple profiles at intervals of a fixed number of instructions, for each application individually. We can then build a single combined profile for the combined instruction stream. Generating this combined profile is dependent on the level of multithreading (coarse- or fine-grained).

5.2.2 Quantification of compiler optimization flags on design space exploration

The quantification of the impact of compiler optimization flags on design space exploration was done using 250 different sets of compiler optimization flags on GNU gcc for Alpha ISAs. Hence, these sets of compiler flags did not include specialized instructions that benefit from hardware acceleration such as security extensions. Therefore it would be interesting to do the study on different ISAs, such as ARM, which provide hardware acceleration for operations such as the SHA hashing algorithm. We expect a much higher impact on design space exploration as the applications could offload large parts of their computation to the hardware accelerator for optimizations that include the specialized instructions, but not for the optimizations without these instructions. This could make two different binaries, implementing the same application, behave completely differently and could hence potentially result in very different design decisions. A possible solution is to consider binaries with specialized instructions as different workloads than binaries without these instructions.

5.2.3 Selecting representative benchmark inputs

The techniques presented in Chapter 4 are evaluated for superscalar in-order cores. It would be interesting to evaluate the techniques on very complex out-of-order cores. It is likely that the techniques need to be extended for cores with more complexity. Possible extensions could be to use more than three sample points in CPI-sampled selection technique or to extend the BBV selection technique to be able to select more than two inputs,

e.g., using clustering of BBVs to find a number of cluster-centers to use as representative inputs.

Second, the techniques are only tested for single-core microprocessors. We expect that multicore processors would need a different approach as they need to be robust for all combinations of different benchmarks and different input sets. E.g., executing two instances of application A on a dual-core processor should be robust for both inputs with a very large memory footprint as well as for inputs with an asymmetric memory footprint. A possible approach is to extend selection techniques for applications on multicore processors, such as [66], by using the technique to select the input datasets as well as the applications. To limit the number of inputs to feed into these selection techniques, the techniques presented in this thesis could be used.

Appendix A

Instruction Profiler

In Chapter 3, we discussed a mechanistic performance model that takes an application profile as input and generates performance numbers as output. The application profile consists of a number of program characteristics that are used by the model's analytical formulas to calculate the individual penalty cycles in each term of Formula 3.1. This appendix discusses our implementation of the profiling tool.

The profiling tool is a hook into gem5's [3] functional simulator. We inspect each instruction in the trace and handle them by the main instrumentation blocks, depending on the type of instruction. The main instrumentation blocks are cache simulators, branch predictors and the H-matrix [1] generator.

The last level cache(s) [2] are simulated with a single-pass cache simulator, based on the algorithms in [58]. This allows us to capture miss rates from cache configurations with a fixed block size, but a different number of sets and different associativity in a single run. To be able to vary cache block sizes we simulate two instances of the single-pass cache simulator (one for 32 byte blocks and one for 64 byte blocks). Lower levels of caches and TLB's, with fixed configurations, are simulated using Dinero [18]. Note that these configurations need to remain fixed since the results (hit or miss) are fed to the next level in the cache hierarchy.

We use gem5's branch predictor units for each branch we encounter in the instruction stream and record the total number of hits and misses for both taken and not taken branches. We instantiate multiple branch predictors in order to be able to get prediction results for different configurations in the same run.

We now explain the generation of the H-matrix using the pseudo-code of Algorithm 1 and the functions it uses in Algorithms 2, 3, 4, 5 and 6.

[1] See Section 3.4 of Chapter 3 for a formal definition and an example H-matrix.

[2] In our experiments we simulated up to two levels of caches.

Algorithm 1 basically declares the global variables and initializes some im-

```
global maxWidth ;
global currPattern ;
global producers ;
global HMatrix ;
maxWidth = 4 ;
currPattern = "XXXX" ;
producers.invalidateAll () ;
HMatrix.init (0) ;
foreach Instruction inst do
    insertInstruction (inst) ;
end
```

Algorithm 1: The main routine of the H-matrix generator.

portant structures before it starts the main profiling. maxWidth is the maximum superscalar width we want to predict performance for. We consider a maxWidth of 4 here to ease the discussion. currPattern contains the current history of instructions up to 4 instructions, as defined by maxWidth. We initialize the pattern with X-instructions (don't care instructions). producers contains a mapping of architectural registers to a structure containing information on the last instruction that produces results for the corresponding register. The information stored in the structure is the sequence number of the instruction (seqNum) and the type of the instruction that produced the result (instType). For initialization we invalidate all the architectural registers. Finally, HMatrix is the matrix of counters that gets updated throughout the instrumentation. We initialize all the counters to 0.

```
Function insertInstruction(inst):
begin
    updatePattern (inst);
    distance = getClosestDependence (inst);
    HMatrix[ currPattern, distance ] += 1 ;
    insertProducer (inst);
end
```

Algorithm 2: The function insertInstruction.

Algorithm 2 describes the functionality of insertInstruction. We first update currPattern by calling updatePattern as we will explain later in Algorithm 3. Next, we determine the distance to the closest instruction producing a result that the current instruction inst needs as input. This is explained in Algorithm 4. We then use this distance together with currPattern to update the corresponding counter in HMatrix. Finally, we update the register-mapping of producers with the current instruction, as explained in

```
Function updatePattern(inst):
    switch inst.instType do
        case intALU
            itype = 'A';
        end
        case intMDU
            itype = 'M';
        end
        case fpALU
            itype = 'F';
        end
        case fpMDU
            itype = 'G';
        end
        case Load
            itype = 'L';
        end
        otherwise
            itype = 'X';
        end
    endsw
    currPattern.pop_front () ;
    currPattern.push_back (itype) ;
```

Algorithm 3: The function updatePattern.

Algorithm 6.

To update the pattern we look at the instruction type of the current instruction, as shown in Algorithm 3. We distinguish five different types of instruction classes: integer ALU instructions (A), integer multiply or divide [3] instructions (M), floating point ALU instructions (F), floating point multiply or divide instructions (G) and load instructions (L). Other instructions, such as stores and branches, which rarely impact the functional unit contention penalty, are not considered here and fall under the don't care category (X). Once the instruction type is identified, we pop the first instruction of currPattern and add the newly identified instruction type to the back.

Algorithm 4 details how to calculate the distance to the closest instruction that produces results required by inst. We loop over all source registers of inst and verify whether producers contains (a) valid instruction(s) pro-

[3] To distinguish between multiplies and divides we also record an instruction mix. This is necessary to distinguish between the difference in latency of multiply versus divide instructions.

```
Function getClosestDependence(inst):
    found = False ;
    for i in inst.numSourceRegs () do
        if ! producers[ inst.sourceReg (i) ].isValid () then
            continue ;
        curDistance = inst.seqNum - producers[ inst.sourceReg (i)
        ].seqNum ;
        curProdType = producers[ inst.sourceReg (i) ].instType ;
        if curDistance < maxWidth or ((curDistance < 2*maxWidth) and
        (isLoad (curProdType) or isLongLatency (curProdType))
        then
            if ! found then
                found = True ;
                distance = curDistance ;
                producerType = curProdType ;
            else if found and curDistance < distance then
                distance = curDistance ;
                producerType = curProdType ;
    end
    if found then
        updateDependenceBarrier (producerType) ;
    else
        distance = 2*maxWidth ;
    return distance
```

Algorithm 4: The function getClosestDependence.

ducing a result for these registers. If the distance to a producer is smaller than maxWidth we set the found-flag. We also set the found-flag if the producer is a load instruction or a long latency instruction (i.e., M-, F- or G-instruction) for distances smaller than 2×maxWidth. If the found-flag was set we also update the dependence barrier, which is shown in Algorithm 5. If the instruction has multiple source registers, we return the shortest distance. If no producer was found at a distance smaller than 2×maxWidth, we return 2×maxWidth as distance. Instructions with a dependence distance of 2×maxWidth are ignored by the Formulas in Chapter 3 as they do not suffer from penalties.

During the time that a dependence gets resolved there are a number of other instructions that produce results. Hence, future instructions, that are dependent on these results will not need to wait. This means that there are dependences that get resolved underneath other dependences and hence this penalty is hidden. We account for this using Algorithm 5. Based on the instruction type of the current producer which introduces a dependence (depInstType) and the instruction type of the other producers, we deter-

```
Function updateDependenceBarrier(depInstType):
    for prod in producers do
        if ! isLongLatency (prod.instType) and! isLoad
        (prod.instType) then
            prod.invalidate ()
        else if isLongLatency (depInstType) and isLoad
        (prod.instType) then
            prod.invalidate ()
    end
```

Algorithm 5: The function updateDependenceBarrier.

```
Function insertProducer(inst):
    if isLongLatency (inst.instType) then
        for prod in producers do
            if ! isLongLatency (prod.instType) then
                prod.invalidate () ;
        end
    for i in inst.numDestRegs () do
        producers[ inst.destReg (i) ].update (inst.seqNum,
        inst.instType) ;
    end
```

Algorithm 6: The function insertProducer.

mine whether or not to invalidate another producer. E.g., integer ALU instructions get invalidated as they are able to produce their result in the same cycle the dependence gets resolved.

The last part of pseudo-code updates the architectural register mapping in producers with the new instruction, as shown in Algorithm 6. It does this for all the destination registers of the new instruction. Before doing so, however, we invalidate other producers, based on the type of instruction that is getting inserted. Similar as with Algorithm 5 this is to account for dependence penalties that get resolved underneath a long latency instructions

Appendix B

Representative Benchmark inputs

In Chapter 4, we presented various selection techniques to find representative benchmark inputs to drive microarchitectural design space exploration. While the techniques are straightforward to implement and incur a low one-time cost only, we present the resulting input data sets in this appendix. In addition, a Bitbucket project [1] has been made that presents the same information on the accompanying wiki, together with links to download the input data sets.

Table B.1 gives a mapping of benchmarks to the directories where their inputs reside after downloading the KDatasets database [2]. In addition, the file extensions are provided in Table B.1. Table B.2 reports the input-numbers that were found by the BBV Selection and CPI Sampled Selection techniques.

Using Table B.1 and Table B.2, we can select the inputs that were found to be representative in our experimental setup. As an example, suppose we wish to use the two inputs found by BBV Selection for dijkstra, we can read from Table B.2 that the inputs with numbers 857 and 40 are representative. From Table B.1, we read that the inputs for dijkstra are located in the "newtwork_dijkstra" directory and have extension ".dat". Therefore the two representative benchmark inputs for dijkstra, according to BBV Selection are network_dijkstra/857.dat and network_dijkstra/40.dat.

Note that the benchmark sha uses a mix of inputs from different categories. Table B.2 distinguishes between these by putting the directory name in parentheses. Further, stringsearch1 requires two input files (the string to

[1] At the time of writing, this project is located at `https://bitbucket.org/maximilien_breughe/representative-benchmark-inputs/wiki/Home`

[2] At the time of writing, the KDatasets website can be found at `http://kdatasets.appspot.com/`

Benchmark	Directory	Extension
adpcm_c	wav	.wav
adpcm_d	adpcm	.adpcm
dijkstra	network_dijkstra	.dat
gsm	au	.au
jpeg_c	ppm	.ppm
jpeg_d	jpg	.jpg
lame	wav	.wav
patricia	network_patricia	.udp
qsort1	automotive_qsort1	.dat
rsynth	txt	.txt
sha	jpg mp3 ps tif txt	.jpg .mp3 .ps .tif .txt
stringsearch1	txt	.txt and .s.txt
susan_c	pgm	.pgm
susan_e	pgm	.pgm
susan_s	pgm	.pgm
tiff2bw	tif	.tif
tiff2rgba	tif	.tif
tiffdither	tif	.tif
tiffmedian	tif	.tif

Table B.1: Representative inputs for MiBench, characterized with the techniques of Chapter 4.

be searched and the source text to search in). The same input number is used for both these files.

Benchmark	BBV Selection	CPI Sampled Selection		
	2 inputs	1 input	2 inputs	3 inputs
adpcm_c	307, 15	318	318, 383	307, 318, 94
adpcm_d	631, 923	376	376, 25	249, 376, 18
dijkstra	857, 40	541	541, 508	3, 541, 856
gsm	13, 552	174	174, 381	880, 174, 9
jpeg_c	11, 5	738	738, 845	13, 738, 870
jpeg_d	13, 22	499	499, 399	71, 499, 22
lame	102, 930	633	633, 489	930, 633, 641
patricia	21, 1000	180	180, 562	1000, 180, 21
qsort1	987, 364	400	400, 336	969, 400, 987
rsynth	559, 916	968	968, 537	916, 968, 587
sha	56 (jpg), 116 (txt)	86 (mp3)	86 (mp3), 492 (txt)	56 (jpg), 86 (mp3), 116 (txt)
stringsearch1	925, 8	416	416, 981	361, 416, 8
susan_c	34, 900	607	607, 528	3, 607, 5
susan_e	919, 34	608	608, 341	34, 608, 919
susan_s	5, 17	295	295, 329	112, 295, 650
tiff2bw	11, 982	249	249, 854	17, 249, 993
tiff2rgba	11, 982	278	278, 271	13, 278, 11
tiffdither	50, 13	794	794, 548	55, 794, 536
tiffmedian	5, 17	211	211, 516	28, 211, 970

Table B.2: Representative inputs for MiBench, characterized with the techniques of Chapter 4.

Bibliography

[1] M. Barr. Real men program in c. *Embedded Systems Design (UBM TechInsights)*, pages 9–12, August 2009.

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

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

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

[5] S. Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference (DAC)*, pages 746–749, June 2007.

[6] P. E. Ceruzzi. *A History of Modern Computing*. MIT Press, 1998.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, October 2009.

[8] X. E. Chen and T. M. Aamodt. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 59–70, December 2008.

[9] X. E. Chen and T. M. Aamodt. A first-order fine-grained multithreaded throughput model. In *Proceedings of the 20th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 329–340, 2009.

[10] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. Evaluating iterative optimization across 1000 datasets. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 448–459, June 2010.

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

[12] R. P. Colwell. *The Pentium Chronicles: The People, Passion, and Politics Behind Intel's Landmark Chips (Software Engineering "Best Practices")*. Wiley-IEEE Computer Society Press, 2005.

[13] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9:256–268, October 1974.

[14] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 266–277, July 2001.

[15] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th ACM International Conference on Computing Frontiers*, CF '07, pages 131–142, 2007.

[16] C. Dubach, T. M. Jones, and M. F. P. O'Boyle. Microarchitecture design space exploration using an architecture-centric approach. In *Proceedings of the IEEE/ACM Annual International Symposium on Microarchitecture (MICRO)*, pages 262–271, December 2007.

[17] C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, and M. F. P. O'Boyle. Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 78–88, 2009.

[18] J. Edler and M. D. Hill. Dinero IV trace-driven uniprocessor cache simulator. Available through http://www.cs.wisc.edu/~markhill/DineroIV, 1998.

[19] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Workload design: Selecting representative program-input pairs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 83–94, September 2002.

[20] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Designing workloads for computer architecture research. *IEEE Computer*, 36(2): 65–71, February 2003.

[21] L. Eeckhout, J. Sampson, and B. Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization (IISWC)*, pages 2–12, October 2005.

[22] S. Eyerman and L. Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. In *The International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–102, March 2010.

[23] S. Eyerman, J. E. Smith, and L. Eeckhout. Characterizing the branch misprediction penalty. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 48–58, March 2006.

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

[25] S. Eyerman, K. Hoste, and L. Eeckhout. Mechanistic-empirical processor performance modeling for constructing CPI stacks on real hardware. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 216–226, April 2011.

[26] M. Ferdman, A. Adileh, Y. O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–48, March 2012.

[27] P. Greenhalgh. Big.little processing with arm cortex-a15 & cortex-a7: Improving energy efficiency in high-performance mobile platforms. http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf, September 2011.

[28] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded

benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization (WWC)*, December 2001.

[29] A. Gutierrez, J. Pusdesris, R.G. Dreslinski, T. Mudge, C. Sudanthi, C.D. Emmons, M. Hayenga, and N. Paver. Sources of error in full-system simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 13–22, March 2014.

[30] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and more flexible program analysis. *Journal of Instruction-Level Parallelism*, 7(11):1571–1580, September 2005.

[31] A. Hartstein and T. R. Puzak. The optimal pipeline depth for a microprocessor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 7–13, May 2002.

[32] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, September 2006.

[33] ARM Holdings. *Cortex-A8: Technical Reference Manual*. ARM Holdings, 110 Fulbourn Road, Cambridge, GB-CB1 9NJ, 3p2 edition, May 2010.

[34] K. Hoste and L. Eeckhout. Cole: Compiler optimization level exploration. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 165–174, 2008.

[35] Texas Instruments Incorporated. *AM335x Sitara Processors: Technical Reference Manual*. Texas Instruments Incorporated, 12500 TI Boulevard, Dallas, Texas 75243, USA, 6 2014.

[36] E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 195–206, October 2006.

[37] Z. Jin and A. C. Cheng. SubsetTrio: An evolutionary, geometric and statistical benchmark subsetting approach. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 21, March 2011.

[38] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 161–170, December 2006.

[39] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. Construction and use of linear regression models for processor performance analysis.

In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 99–108, February 2006.

[40] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. *IEEE Transactions on Computers*, 55(6):769–782, June 2006.

[41] T. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 338–349, June 2004.

[42] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1(2):10–13, June 2002.

[43] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 236–247, March 2005.

[44] H. Leather, E. Bonilla, and M. O'boyle. Automatic feature generation for machine learning–based optimising compilation. *ACM Transactions on Archititecture Code Optimization (TACO)*, 11(1):14:1–14:32, February 2014.

[45] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 185–194, October 2006.

[46] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 469–480, December 2009.

[47] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 190–200, June 2005.

[48] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 2–10, October 1999.

[49] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 89–100, June 2007.

[50] E. Ould-Ahmed-Vall, J. Woodlee, C. Yount, K. A. Doshi, and S. Abraham. Using model trees for computer architecture performance analysis of software applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 116–125, April 2007.

[51] E. Park, S. Kulkarni, and J. Cavazos. An evaluation of different modeling techniques for iterative compilation. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 65–74, October 2011.

[52] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 412–423, June 2007.

[53] R. N. Sanchez, J. N. Amaral, D. Szafron, M. Pirvu, and M. Stoodley. Using machines to learn method-specific compilation strategies. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 257–266, April 2011.

[54] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer. Modeling performance variation due to cache sharing. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 155–166, February 2013.

[55] Y. Sazeides, R. Kumar, D. M. Tullsen, and T. Constantinou. The danger of interval-based power efficiency metrics: When worst is best. *IEEE Computer Architecture Letters*, 4, 2005.

[56] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 3–14, September 2001.

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

[58] R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Transactions on Computer Systems (TOCS)*, 13(1):32–56, February 1995.

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

[60] M. Tartara and S. Crespi R. Continuous learning of compiler heuristics. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9 (4):46:1–46:25, January 2013.

[61] M. Van Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 45–56, March 2004.

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

[63] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Representative multiprogram workloads for multithreaded processor simulation. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 193–203, October 2007.

[64] K. Van Craeynest and L. Eeckhout. The multi-program performance model: Debunking current practice in multi-core simulation. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 26–37, November 2011.

[65] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 213–224, June 2012.

[66] R. A. Velasquez, P. Michaud, and A. Seznec. Selecting benchmark combinations for the evaluation of multicore throughput. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2013.

[67] Xiph.org. Video test media - derf's collection. `http://media.xiph.org/video/derf/`, 2012.

[68] J. J. Yi, D. J. Lilja, and D. M. Hawkins. A statistically rigorous approach for improving simulation methodology. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA)*, pages 281–291, February 2003.

[69] 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 *Proceedings of the 20th ACM International Conference on Supercomputing (ICS)*, pages 75–86, June 2006.