Microarchitectuur-onafhankelijke analytische prestatie- en vermogensmodellering van processors

Microarchitecture-Independent Analytical Processor Performance and Power Modeling

Sam Van den Steen

Promotor: prof. dr. ir. L. Eeckhout
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: computerwetenschappen

UNIVERSITEIT
GENT

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. K. De Bosschere
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2018 - 2019

ISBN 978-94-6355-155-7
NUR 987
Wettelijk depot: D/2018/10.500/73

Examination Committee

Prof. dr. ir. Filip De Turck, *voorzitter*
Vakgroep INTEC, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

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

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

Prof. dr. ir. Bart Dhoedt
Vakgroep INTEC, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. dr. Erik Hagersten
Department of Information Technology
Uppsala Universitet

Dr. ir. Stijn Eyerman
Intel Belgium

Dr. Josué Feliu Pérez
Parallel Architectures Group
Universitat Politècnica de València

Contents

Examination Committee **i**

Dankwoord **vii**

Samenvatting **ix**

Summary **xiii**

List of Figures **xvii**

List of Tables **xxi**

List of Abbreviations **xxiii**

1 Introduction **1**

1.1 Motivation 1

1.2 Key Contributions 2

1.2.1 Micro-architectural Independent Analytical Model . . . 2

1.2.2 Sampled Model Evaluation 3

1.2.3 Modeling Memory-Level Parallelism 4

1.2.4 Open Sourced Framework 4

1.3 Other Research Activities 5

1.4 Thesis Overview 5

2 Background **7**

2.1 Out-of-order Processor 7

2.2 Architectural Simulation 10

2.2.1 Timing Simulation versus Functional Simulation 10

2.2.2 Sampled Simulation 12
2.3 Performance Modeling 13
2.3.1 Empirical Modeling 13
2.3.2 Mechanistic Modeling 14
2.4 Power Modeling 15
2.5 Interval Model 16
2.5.1 I-Cache Misses 17
2.5.2 Branch Mispredictions 18
2.5.3 D-Cache Misses 20
2.5.4 Interval Model Equation 21
2.6 Micro-Architecture Independent Modeling 22

3 Modeling the Core **25**
3.1 Improvements to the Interval Model 25
3.2 CISC versus RISC 26
3.3 Instruction Dependences 27
3.4 Issue Stage Modeling 33
3.5 Branch Predictor Modeling 38
3.6 Core Power Modeling 42

4 Modeling the Memory Subsystem **45**
4.1 Cache Hierarchy and the Interval Model 45
4.2 Cache Miss Rate Modeling 46
4.3 Memory-Level Parallelism 49
4.4 Cold-Miss MLP Model 52
4.5 Stride-MLP Model 54
4.6 Modeling MSHRs 57
4.7 Main Memory Bus 59
4.8 Chained LLC Hits 60
4.9 Hardware Prefetching 62
4.10 Memory Power Modeling 64

5 Sampling Methodology **65**
5.1 Instruction Mix Sampling 65
5.2 Dependence Chain Interpolation 67

5.3 Dependence Chain Sampling 69
5.4 Memory Sampling 70
5.4.1 Cache Miss Rates 70
5.4.2 Memory-Level Parallelism 71

6 Evaluation 73
6.1 Experimental Setup 73
6.2 Performance Prediction 74
6.2.1 Absolute Accuracy 74
6.2.2 Sampled Profiling 76
6.2.3 Micro-Architecture Independent Modeling 78
6.2.4 Relative Accuracy across a Design Space 79
6.3 Power Prediction 83
6.3.1 Absolute Accuracy 83
6.3.2 Relative Accuracy across a Design Space 84
6.4 CPI Stacks 87
6.5 Phase Analysis 90
6.6 Stride MLP and Prefetch Results 92

7 Applications 97
7.1 Understanding Processor Performance 97
7.2 Optimizing Performance under Power Constraints 99
7.3 DVFS Exploration 101
7.4 Pareto Curves 102
7.5 Comparison to an Empirical Model 108

8 Conclusion 113
8.1 Summary 113
8.2 Future Work 116
8.2.1 Multi-core Processors 116
8.2.2 SMT Processors 116
8.2.3 Multithreaded Workloads 117
8.2.4 Different ISAs 117

Appendices **119**

A CPI stacks 119

B Phase plots 127

C Pareto plots 139

Bibliography **149**

Dankwoord

Het uitvoeren van een doctoraatsonderzoek is een werk van lange adem. Gedurende een viertal jaar werk je grotendeels op één onderwerp om daar dan uiteindelijk een thesis over te schrijven. Bij momenten is de inspiratie ver te zoeken en heb je een grote dosis doorzettingsvermogen nodig. In dit dankwoord wil ik alle mensen bedanken die hebben meegeholpen om mijn doctoraat succesvol te beëindigen.

Eerst en vooral wil ik mijn promotor Lieven Eeckhout bedanken. Zonder hem was ik uiteraard nooit aan dit doctoraat kunnen starten. Voorts was zijn ervaring met het schrijven van papers onmisbaar. Zijn kritische ingesteldheid tijdens de vele *face-to-face meetings* dwongen mij om alle problemen van verschillende kanten te bekijken en te analyseren. Dit heeft heel veel bijgedragen tot de uiteindelijke kwaliteit van dit werk en uiteraard tot mijn ontwikkeling als onderzoeker. Verder wil ik ook Stijn Eyerman nog persoonlijk bedanken. Zijn continue begeleiding gedurende het eerste anderhalf jaar hebben ervoor gezorgd dat ik een vliegende start heb kunnen nemen met dit doctoraat.

I would like to thank all members of my jury. The internal defense that took place before finishing this thesis was, surprisingly, a fun experience. Furthermore, it is because of their critical questions and insightful feedback that I was able to further improve this thesis.

Uiteraard wil ik ook mijn vriendin, Helena, bedanken voor al haar steun de afgelopen jaren. Hoewel mijn doctoraat niet eenvoudig uit te leggen valt, heb ik toch meerdere keren pogingen mogen doen. Hierdoor kan ik haar tegenwoordig al wel eens aan het woord laten wanneer er iemand vraagt om mijn doctoraat uit te leggen, wat best handig is.

Vervolgens wil ik ook mijn ouders, broer en zus bedanken voor hun steun tijdens mijn studies en dit doctoraat. Er zijn momenten geweest dat mijn ouders een beetje skeptisch waren over mijn speciaal studietraject, maar gelukkig was ik koppig genoeg om het allemaal tot een goed einde te brengen.

Ten slotte wil ik uiteraard al mijn labogenoten bedanken. Gedurende de afgelopen jaren heb ik voornamelijk samengewerkt met Sander en vele discussies met hem gevoerd over alle uiteenlopende modellen in dit werk. Ook wil ik nog een aantal andere collega's en ex-collega's bedanken voor de ontspannende gesprekken en occasionele discussies: Ajeya, Cecilia, Jennifer, Josué, Kartik, Kristof, Mutaz, Shoaib, Trevor, Wim, Xia. Verder wil ik Bart nog bedanken voor de vele ontspannende roddels op momenten dat de zin om te werken net iets verder te zoeken was. Indien ik iemand vergeten te bedanken ben, wil dat uiteraard niets meer zeggen dan dat ik slecht ben in lijstjes maken.

Sam Van den Steen
Gent, September 26, 2018

Samenvatting

Tot in het begin van de jaren 2000 focusten processorarchitecten zich voornamelijk op het versnellen van processors. Ze slaagden hierin door processors te voorzien van steeds complexere structuren zodat de software sneller uitgevoerd kon worden. Dit was mogelijk door de transistors – de bouwstenen van een processor – steeds verder te miniaturiseren. Hierdoor slaagden ze erin steeds meer transistors op het chipoppervlak te plaatsen en toch nog altijd constante vermogensdensiteit te garanderen. Echter, deze schalingstrend bekend als Dennard-schaling, bleek onmogelijk vol te houden.

Ondertussen explodeerde, door het ontstaan van het digitale tijdperk, het aantal mobiele toestellen met één of meerdere processors. Tegenwoordig wordt de meerderheid van processors, bedoeld voor de normale gebruikers, ingebouwd in mobiele toestellen zoals GSM's of *tablets*. Deze toestellen werken door middel van batterijen waardoor de processor best zo weinig mogelijk vermogen verbruikt en zo weinig mogelijk warmte genereert. Ondertussen moeten ze nog steeds voldoende prestatie bieden zodat de eindgebruiker niet de indruk heeft dat zijn/haar toestel 'traag' is. Niet enkel processors van mobiele toestellen ondervonden het nadeel van de stijgende vermogensdensiteit, zelfs processors in servers genereren tegenwoordig dusdanig veel warmte dat deze nog moeilijk efficiënt kan afgevoerd worden. Hierdoor moesten processorarchitecten hun focus op de prestatie van een processor (deels) verleggen naar een focus op energie- en vermogensefficiëntie.

De hoofdvraag voor processorarchitecten werd dus hoe de energie- en vermogensefficiëntie van een processor te verbeteren. Helaas is het fysiek produceren van een nieuw type processor om deze te evalueren tijdens het ontwerpsproces onmogelijk aangezien dit extreem duur en bijzonder tijdrovend is. Hedendaagse processors zijn bovendien heel complex en bestaan uit miljarden transistors waardoor het moeilijk is om ze te optimalizeren. Daarom vertrouwen processorarchitecten doorgaans op softwaresimulatie om nieuwe processors te evalueren tijdens de ontwerpsfase. Helaas zijn deze simulaties meerdere grootteordes trager dan het uitvoeren op een echte processor. Bovendien houdt de complexiteit van een processor in dat processorarchitecten doorgaans een ontwerpsruimte van processors moeten evalueren aangezien het effect van een optimalisatie niet altijd onmiddellijk duidelijk is. Dit leidt tot het probleem waarbij het ontwerpen van een processor, en dus de tijd tot deze

geïntroduceerd wordt op de markt, vaak meerdere jaren in beslag neemt. Gezien de sterke concurrentie in deze markt, is dit onwenselijk.

Een alternatieve oplossing voor het evalueren van de prestatie van een processor en zijn vermogensverbruik is het inzetten van een mechanistisch analytisch model. Een mechanistisch model modelleert de eerste-orde interacties tussen een applicatie en de processor waarop deze uitvoert. Deze methodologie heeft als voordeel dat ze meerdere grootteordes sneller is dan simulatie, maar ook nog steeds vrij nauwkeurige resultaten produceert. Het gebruik van een mechanistisch analytisch model behelst doorgaans twee stappen: een profileringsstap (dit is de traagste stap) waarin applicatiekarakteristieken worden verzameld en een analysestap (die doorgaans slechts enkele seconden duurt) om de prestatie en het vermogensverbruik te voorspellen.

Het nadeel van de eerder voorgestelde mechanistische modellen is dat ze afhankelijk zijn van verschillende functionele simulaties om alle inputs te verzamelen. Hieronder vallen, onder meer, simulaties om het aantal incorrect voorspelde sprongen (sprongmissers) te bepalen, maar ook het aantal cachemissers en het aantal parallelle toegangen naar het hoofdgeheugen (Memory-Level Parallelism of MLP). Ook al zijn deze functionele simulaties veel sneller dan volledige tijdsgetrouwe simulaties, dewelke processorarchitecten normalerwijze gebruiken, toch introduceren ze nog steeds een niet-verwaarloosbare vertraging bij het evalueren van een grote processorontwerpsruimte.

Deze thesis pakt daarom twee problemen aan. Allereerst willen we hedendaagse, superscalaire, *out-of-order* x86-processors modelleren. Ten tweede willen we het gebruik van meerdere functionele simulaties om de inputs te verzamelen elimineren en zo het evalueren van een ontwerpsruimte versnellen. Daarom stellen we een nieuw, micro-architecturaal onafhankelijk, mechanistisch model voor dat zowel de prestatie als het vermogensverbruik van een processor kan voorspellen. Het grote voordeel van deze methodologie is dat de traagste stap, het verzamelen van het applicatieprofiel, slechts één keer moet uitgevoerd worden. Dit applicatieprofiel kan vervolgens gebruikt worden om er de inputs voor het analytisch model uit af te leiden en de prestatie en het vermogensverbruik van een waaier aan processors te voorspellen.

Om een mechanistisch model te ontwikkelen dat in staat is hedendaagse x86-processors te modelleren, vertrekken we van het eerder voorgestelde intervalmodel [32]. Eerst stellen we een aantal aanpassingen voor aan de basiscomponent die de maximaal haalbare prestatie voorspelt wanneer er geen sprong- of cachemissers zijn. Het eerder voorgestelde intervalmodel maakte gebruik van instructies als kleinste werkeenheid, maar om x86-processors te modelleren, moeten we dit vervangen door het aantal micro-operaties afgeleid van de dynamische instructiestroom. Bovendien delen we het aantal micro-operaties door de effectieve *dispatch*-snelheid, in plaats van deze te delen door de fysieke *dispatch*-breedte. *Dispatch* refereert hier naar de pijplijnstap waarbij instructies van de *front-end* van de pijplijn naar de *back-end* gestuurd worden. De effectieve *dispatch*-snelheid modelleert contentie in de processor ten gevolge van een (deels) ongebalanceerde processorpijplijn. Ze modelleert contentie door afhankelijkheden in de instructiestroom en contentie in de *issue*-stap in de processor ten gevolge van de functionele eenheden en *issue*-poorten. Voorts tonen we aan dat het modelleren van contentie in de processorkern vereist dat ons mechanistisch model wordt geëvalueerd op een zeer kleine tijdsschaal met behulp van wat wij *micro-traces* noemen. Dit is intuïtief logisch aangezien contentie doorgaans in bursts optreedt en op een grote tijdsschaal enkel uitgemiddeld gedrag gemodelleerd kan worden.

Voorts introduceren we extra modelleringsstappen die begrenzingen met betrekking tot geheugentoegangen modelleren. We stellen voor om de impact van *miss status handling registers (MSHR)* te modelleren en voegen een component toe die de extra vertragingen geïntroduceerd door het parallel uitvoeren van geheugentoegangen modelleert. Bovendien introduceren we een nieuwe vertragingscomponent die van elkaar afhankelijke toegangen naar het laatste cacheniveau modelleert. Deze term is vereist omdat een *out-of-order* processor normaal de tijd die gewacht moet worden op data kan verbergen door andere instructies uit te voeren. Echter, het laatste niveau van de cachehiërarchie is dusdanig traag dat deze veronderstelling niet altijd opgaat, zeker wanneer meerdere toegangen naar dit laatste cacheniveau afhankelijk zijn van elkaar.

Naast het aanpassen van het eerder voorgestelde intervalmodel om nauwkeurig prestatie en vermogensverbruik van een x86-processor te voorspellen, was het ook de bedoeling om de inputs verkregen met behulp van functionele simulatie te vervangen door inputs berekend uit micro-architectuur onafhankelijke statistieken. Daarom moeten we het aantal sprongmissers, cachemissers en MLP kunnen voorspellen. Het aantal sprongmissers voorspellen we aan de hand van een metriek die lineaire sprongentropie genoemd wordt. Deze metriek modelleert de (on)voorspelbaarheid van spronginstructies en laat toe de nauwkeurigheid van een sprongvoorspeller te schatten [22]. Het voorspellen van cachemissers gebeurt door het profileren van een distributie van hergebruiksafstanden. Deze hergebruiksafstanden kunnen, door gebruik te maken van StatStack [28], omgevormd worden tot een distributie van *stack distances*. Het voordeel van *stack distances* is dat deze gebruikt kunnen worden om cachemissers te voorspellen voor *least-recently used (LRU)* cachehiërarchieën. Het modelleren van MLP bleek één van de moeilijkste aanpassingen te zijn omdat dit afhankelijk is van verschillende factoren. We stellen twee methoden voor, het *cold-miss MLP* model en het *stride-MLP* model, die steunen op verschillende veronderstellingen gerelateerd aan het burstgedrag van geheugentoegangen.

Gebruik makend van de voorgestelde aanpassingen en inputs afgeleid uit micro-architectuur onafhankelijke statistieken voorspellen we de prestatie en het vermogensverbruik van een referentie processorarchitectuur. De gemiddelde voorspellingsfouten zijn slechts 7.6% en 3.4% voor, respectievelijk, de prestatie en het vermogensverbruik in vergelijking tot cyclus-getrouwe simulaties. Ook tonen we hoe het model gebruikt kan worden om CPI-stapels te bouwen. Deze stapels zijn zeer nuttig om te analyseren waaraan de uitvoeringstijd van een applicatie gespendeerd wordt op een bepaalde processor.

Het hoofddoel van een micro-architectuur onafhankelijk mechanistich model is het versnellen van de evaluatie van een ontwerpsruimte. Gebruik makend van dit model kunnen we een ontwerpsruimte, bestaand uit 243 processors en 29 applicaties, evalueren in 11,5 uur. Dezelfde ontwerpsruimte evalueren met behulp van simulaties, die uitvoeren aan een snelheid van 0,5 miljoen instructies per seconde, (MIPS) zou 150 dagen in beslag nemen. Het eerder voorgestelde intevalmodel kan de ontwerpsruimte evalueren in 200 uur ervanuit gaande dat de functionele simulaties een snelheid van 1,5 MIPS halen. Dit betekent dat we een versnelling van 315× behalen ten opzichte van gedetaileerde simulatie en 18× vergeleken ten opzichte van het eerder voorgestelde intervalmodel. Indien we onze voorspellingen met betrekking tot de prestatie en het vermogensverbruik voor de volledige ontwerpsruimte evalueren, behalen we een nauwkeurigheid van respectievelijk 9.3% en 4.3% voor prestatie- en vermogensvoorspellingen.

Ten slotte tonen we de bruikbaarheid van het model aan door Paretofronts te construeren die de afweging visualiseren tussen prestatie en vermogensverbruik en daardoor gebruikt kunnen worden om interessante processorarchitecturen uit een ontwerpsruimte te selecteren. Deze toepassing illustreert de belangrijkste eigenschap van ons model, namelijk de relatieve nauwkeurigheid. Deze eigenschap maakt het immers mogelijk om verschillende processors met elkaar te vergelijken, ongeacht de eventuele absolute fout op de prestatie- en/of vermogensvoorspellingen. We vatten de nauwkeurigheid van het filteren van de ontwerpsruimte samen met behulp van vier metrieken: sensitiviteit, specificiteit, nauwkeurigheid en HVR. Deze metrieken geven weer hoe goed we erin slagen Pareto-optimale ontwerpen te selecteren over de volledige ontwerpsruimte. De gemiddelde waardes voor sensitiviteit, specificiteit, nauwekeurigheid en HVR zijn respectievelijk 46.2%, 87.9%, 76.8% en 97.0%. De nauwkeurige specificiteitswaarde toont aan dat we erin slagen om de meeste niet-Pareto optimale architecturen weg te filteren terwijl de HVR-metriek aantoont dat we architecturen over de volledige ontwerpsruimte vinden. De gemiddelde waarde voor sensitiviteit is relatief laag, wat erop wijst dat we niet alle Pareto-optimale architecturen vinden. Dit is echter geen groot probleem omdat Pareto-optimale ontwerpen vaak voorkomen in clusters en één van die ontwerpen vinden reeds voldoende is. Globaal genomen toont dit aan dat we er in slagen om een ontwerpsruimte te exploreren en de Pareto-optimale ontwerpspunten te identificeren.

Deze thesis vat het werk samen ter ontwikkeling van een micro-architectuur onafhankelijk model voor hedendaagse x86-processors dat in staat is om, in vergelijking met cyclus-getrouwe simulatie, de prestatie en het vermogensverbruik van een processor nauwkeurig te voorspellen. We tonen verder aan dat dit model een significante versnelling behaalt ten opzichte van zowel gedetailleerde simulatie als eerder voorgestelde mechanistische modellen. Dit maakt het mogelijk om grote ontwerpsruimtes van processors te evalueren en interessante processors te identificeren.

Summary

Up until the early 2000's, processor architects focused mainly on developing faster processors. They achieved this through implementing increasingly complex structures into processors to optimize application execution. This was feasible because they could miniaturize transistors – the processor's building blocks – and include more of them on the same processor chip area while still maintaining constant power density. However, this scaling trend, known as Dennard scaling, started to break down shortly afterwards.

Meanwhile, the advent of the digital age introduced an incredible increase in mobile devices containing one or more processors. Nowadays, a majority of processors meant for the consumer market are built into mobile devices such as cell phones or tablets. These devices run on a battery necessitating the processor to consume as little power and generate as little heat as possible while still achieving sufficient performance for the end-user not to experience his/her device as being 'slow'. However, the problem is not limited to processors embedded in mobile devices, due to the increasing power density, even processors in high-end systems generate too much heat to dissipate easily. This required processor architects to not only focus on processor performance, but also on energy and power efficiency.

The main question became how to improve the energy efficiency of a processor. Unfortunately, physically producing new processor prototypes to evaluate during the design cycle is infeasible because this would be extremely expensive and time-consuming. Contemporary processors are also incredibly complex consisting of billions of transistors making it difficult to design and optimize. Therefore, processor architects generally rely on software simulation to design new processors. However, processor simulations are multiple orders of magnitude slower compared to a real execution on a processor. Furthermore, since contemporary processors are so complex, a processor architect will often need to evaluate a design space consisting of multiple different processors, as the effect of an optimization might not be obvious from the start. This introduces the problem where the processor design cycle, and thus time-to-market for a new processor often encompasses several years. Because of the fierce competition in the market of processors, this is undesirable.

An alternative to evaluating processor performance and power consumption using simulation is mechanistic analytical modeling. A mechanistic model models the first-order interactions between the application and the processor it is executing on. The approach of modeling rather than simulating a processor has the advantage that it is orders of magnitude faster while still achieving relatively good accuracy. Employing a model to predict performance and power usually consists of two steps, a profiling phase (which is the slowest step) to collect application characteristics and an analysis phase (which takes only a couple of seconds) to predict performance and power.

The downside of previously proposed mechanistic models is that they rely on several functional simulations to collect the required inputs such as the number of branch mispredictions, cache miss rates, and memory-level parallelism (MLP). While these functional simulations are significantly faster than the full-blown timing-based simulations generally employed by processor architects, they still incur a significant slowdown for evaluating a large processor design space.

This thesis consists of two main objectives. Firstly, we want to model contemporary superscalar out-of-order x86-based processors. Secondly, we want to eliminate the use of several functional simulations to obtain inputs for the mechanistic model in order to speed up design space exploration. Therefore, we propose a new micro-architecture independent mechanistic model to predict both performance and power consumption. The key advantage of this approach is that the slowest step, collecting the application profile, only has to be performed once. This application profile can then be used to predict the inputs to the analytical model for predicting processor performance and power consumption.

To develop a mechanistic model capable of modeling contemporary x86-based processors, we modify the previously proposed interval model [32]. To achieve this, we modify the base component which predicts the maximum achievable performance in the absence of miss events. The previously proposed interval model used instructions as smallest unit of work, but to model x86-processors, we have to replace this with the number of micro-operations derived from the dynamic instruction stream. Furthermore, instead of dividing the number of micro-operations by the physical dispatch width, we introduce a new divisor called the effective dispatch rate. The effective dispatch rate models contention within the processor due to imbalances in the processor pipeline. It models contention due to dependences within the instruction stream and contention due to the functional units and issue ports in the issue stage. We also show that modeling contention in the processor core requires evaluating our mechanistic model on very small time scales using what we call micro-traces. Intuitively, this makes sense as contention will mostly occur due to bursty behavior and large time scales are only suitable to capture averaged-out behavior.

We also introduce extra modeling steps to capture limitations related to the memory requests performed by the processor. We propose to model the impact of miss status handling registers (MSHR) and add a component modeling the extra queuing delay introduced by executing parallel accesses to main memory.

Furthermore, we introduce a new type of penalty called last-level cache (LLC) chaining. While out-of-order processors can usually hide the latency of load instructions fetching data from the cache hierarchy, the last level of the cache hierarchy cannot always provide data fast enough for the processor to hide its latency, especially when multiple loads depend on each other.

Next to modifying the previously proposed interval model to accurately predict performance and power for x86-processors, we also replace the inputs extracted from the functional simulations by inputs calculated using micro-architecture independent inputs. To achieve this, we have to predict the number of branch mispredictions, cache miss rates and MLP. Predicting the number of branch mispredictions is achieved through a metric called linear branch entropy which captures the (un)predictability of branch instructions [22]. For predicting cache miss rates we collect a reuse distance distribution which is transformed into a stack distance distribution using StatStack [28]. We can use the latter distribution to predict cache miss rates for least-recently used (LRU) cache hierarchies. Modeling MLP accurately proved to be the most difficult hurdle to overcome as it depends on various factors. We propose two different techniques called the cold-miss MLP and stride-MLP which leverage different assumptions related to the burstiness of memory accesses.

Using the proposed modifications and inputs derived from micro-architecture independent metrics we predict the performance and power consumption for a reference processor design. The average prediction errors are as low as 7.6% and 3.4% for performance and power predictions, respectively, compared to cycle-level simulation. We show the convenience of the model by generating so-called CPI stacks. These stacks are extremely useful to analyze where the cycles go when executing an application on a specific processor design.

The main goal of developing a micro-architectural independent mechanistic model is to speed up design space evaluation. Using the new model we can evaluate a design space of 243 processor architectures and 29 applications in 11.5 hours. Evaluating the same design space using detailed simulation running at 0.5 million instructions per second (MIPS) would take 150 days, while the previously proposed interval model would take 200 hours assuming the functional simulations can progress at a speed of 1.5 MIPS. This means we achieve a speedup of 315× compared to detailed simulation and 18× compared to the previously proposed interval model. Comparing our predictions to cycle-level accurate simulation, we achieve an average prediction accuracy of 9.3% and 4.3% for performance and power, respectively, across the large processor design space considered in this thesis.

We demonstrate the usefulness of the developed micro-architectural independent model by constructing Pareto frontiers visualizing performance-power trade-offs and pruning the design space for interesting designs. This application shows the most important characteristic of our model, namely its relative accuracy. After all, even if the absolute prediction error is significant, as long as all errors across different processor designs exhibit the same bias, it is possible to

accurately compare multiple different designs. We summarize the accuracy of pruning the design space exploration using the sensitivity, specificity, accuracy and HVR metrics. These metrics describe how well we can predict the actual Pareto-optimal processor architectures and whether we are able to find designs of interest over the complete range of the design space. The average values for sensitivity, specificity, accuracy and HVR are 46.2%, 87.9%, 76.8% and 97.0%, respectively. The good specificity value shows that we are able to exclude most non-Pareto optimal designs while the HVR metric shows that we find designs over the complete design space. The average sensitivity value is rather low, indicating we do not find all designs that are Pareto-optimal, but this is less of a problem as many Pareto-optimal processor designs are clustered and finding only one of those is acceptable. Overall, the Pareto-plots and these metrics indicate we are able to prune the design space accurately.

To summarize, this work describes a micro-architecture independent model for contemporary x86-based processors with good relative performance and power prediction accuracy compared to cycle-level simulations. Furthermore, we show that this model offers a significant speedup over simulation and previously proposed mechanistic models when evaluating a processor design space and identifying interesting processor designs.

List of Figures

2.1 Schematic overview of the pipeline of an out-of-order processor. 8
2.2 Program execution according to the interval model. 17
2.3 Interval breakdown of an instruction cache miss. 18
2.4 Interval breakdown of a branch misprediction. 19
2.5 Interval breakdown of a long-latency load data cache miss. . . . 20
2.6 Schematic overview of the micro-architecture independent interval model. 22

3.1 Number of micro-operations per instruction for all SPEC CPU 2006 benchmarks. 27
3.2 Example of a data dependence graph. 28
3.3 Example of the calculation of instruction dependence chains. . 31
3.4 Instruction dependences: average path, average branch path and critical path. 32
3.5 Example of the Nehalem issue stage. 34
3.6 Visualization of the factors limiting the effective dispatch rate. 36
3.7 Decrease in prediction error due to the different limitations for the effective dispatch rate. 37
3.8 Visualization of the training step and application profiling to obtain branch misprediction rates. 40
3.9 Linear fit for branch entropy and missprediction rates for more than 400 experiments. 41
3.10 Box-and-whiskers plot for five different branch predictors showing the prediction accuracy. 41

4.1 Memory address stream with indicated reuses and respective stack distances. 47

4.2 Estimated cache misses for a three-level cache hierarchy with sizes 32, 256 and 8192 KB. 48

4.3 Normalized execution time with and without MLP modeling. . 50

4.4 Breakdown of cold and capacity load and store misses. 51

4.5 Example of a load dependence distribution. 53

4.6 Illustrative virtual memory access stream. 54

4.7 Ratio of all stride categories per benchmark. 56

4.8 Visualization of the queuing delay due to multiple concurrent memory accesses. 59

4.9 CPI variation over time for `gcc` with and without the LLC hit chaining component. 61

4.10 Example of the workings of a stride prefetcher. 63

5.1 Visualization of the micro-traces and windows in an instruction stream. 66

5.2 Comparison of the sampled and non-sampled instruction mix. . 66

5.3 Comparison between profiled and interpolated AP, ABP and CP dependence chains for *astar*. 68

5.4 Error on the dependence chain lengths due to interpolation between ROB sizes 68

5.5 Error on the dependence chain lengths due to sampling. 69

5.6 Relative contribution of the branch component to the total execution time. 70

6.1 CPI stacks generated by the model and by Sniper. 75

6.2 Visualization of the different sampling approaches to collect micro-traces within windows in an instruction stream. 76

6.3 Average absolute prediction error versus number of instructions profiled. 77

6.4 Cumulative distribution of the performance prediction error when evaluating micro-traces separately versus combined. . . . 81

6.5 Box-and-whiskers plot showing the performance prediction errors across our design space. 81

6.6 Micro-architecture independent interval model CPI estimation versus simulated CPI. 82

6.7 Power stacks generated by the model and by Sniper. 83

6.8 Cumulative error distribution for power predictions. 85

6.9 Box-and-whiskers plot showing the power prediction errors across our design space. 85

6.10 Micro-architecture independent interval model power estimation versus simulated power. 86

6.11 Comparison of the base component for the *gamess* and *gromacs* benchmarks. 88

6.12 Comparison of the DRAM component for the *milc* and *mcf* benchmarks. 89

6.13 Comparison of the performance of the *gromacs* benchmark on our reference architecture and a low-power architecture. 90

6.14 Phase graphs for the *astar*, *bzip2* and *cactusADM* benchmarks. 91

6.15 Absolute error for the cold-miss and stride MLP models, assuming no hardware prefetching. 93

6.16 Absolute error for predicting performance using the stride and cold-miss MLP models. 94

6.17 Cumulative error distribution when using the stride MLP model versus the cold-miss MLP model. 95

6.18 Absolute error for the stride and cold-miss MLP models, assuming hardware stride prefetching. 95

7.1 Improving performance for the *libquantum* benchmark. 98

7.2 Average CPI when selecting a general-purpose core versus selecting an application specific core. 99

7.3 ED^{2}P for different frequencies calculated with Sniper and our model. 102

7.4 Pareto frontiers for the *bzip2* and *calculix* benchmarks. 103

7.5 Pareto frontiers for the *gromacs* and *xalancbmk* benchmarks. . 104

7.6 Average absolute error for performance and power predictions for the complete design space. 105

7.7 Sensitivity, specificity and accuracy of the Pareto-filtered predictions on the complete design space. 106

7.8 Illustration of the HVR metric calculation. 106

7.9 HVR of the Pareto-filtered predictions on the complete design space. 107

7.10 Pareto fronts for different benchmarks as calculated by our mechanistic model and the empirical model. 109

7.11 Comparison between the interval model and the mechanistic model of the sensitivity values. 110

7.12 Comparison between the interval model and the mechanistic model for the specificity values. 110
7.13 Comparison between the interval model and the mechanistic model for the HVR values. 111

List of Tables

2.1 Example of an instruction mix of an application. 23

3.1 Two examples of instruction mixes resulting in limitations on the effective dispatch rate. 35

6.1 Core configuration for our reference architecture, based on the Intel Nehalem processor. 74

6.2 Average and maximum error of introducing a new micro-architecture independent component. 78

6.3 Core configuration design space. 79

6.4 Reference architecture, based on Intel Nehalem. 93

7.1 Results of optimizing performance under power constraints. . . 100

7.2 Nehalem-based architecture using different DVFS settings. . . . 101

List of Abbreviations

ABP	Average Branch Path
ALU	Arithmetic Logic Unit
AP	Average Path
ASIP	Application-Specific Instruction Processor
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal Oxide Semiconductor
CPI	Cycles Per Instruction
CPU	Central Processing Unit
CP	Critical Path
DRAM	Dynamic Random Access Memory
DVFS	Dynamic Voltage and Frequency Scaling
ED^2P	Energy-Delay-Square Product
EDP	Energy-Delay Product
EX	Execution
FPU	Floating Point Unit
HVR	Hyper Volume Ratio
ID	Instruction Decode
IF	Instruction Fetch
ILP	Instruction-Level Parallelism
IPC	Instructions Per Cycle
ISA	Instruction-Set Architecture
ITRS	International Technology Roadmap for Semiconductors

L1D	Level-1 Data Cache
L1I	Level-1 Instruction Cache
L2	Level-2 Cache
L3	Level-3 Cache
LLC	Last-Level Cache
LRU	Least Recently Used
MEM	Memory Access
MIPS	Million Instructions Per Second
MLP	Memory-Level Parallelism
MPKI	Misses Per Kilo Instructions
MSHR	Miss Status Handling Registers
OF	Operand Fetch
PAC	Phase-Accuracy Coefficient
RAW	Read-after-Write
RD	Reuse Distance
RISC	Reduced Instruction Set Computer
ROB	Re-Order Buffer
RTL	Register Transfer Level
SD	Stack Distance
SMT	Simultaneous Multithreading
TLB	Translation Look-aside Buffer
WAR	Write-after-Read
WAW	Write-after-Write
WB	Write-back

Chapter 1

Introduction

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

– Albert Einstein

1.1 Motivation

The demise of Dennard scaling, which promised constant power density with every successive process technology [25], ended an era where processor performance improvements originated largely from scaling chip technology. Before its demise, every new processor generation introduced performance gains because transistors were miniaturized, could switch faster and consumed less energy. Hence, processor designers could integrate more transistors on a chip realizing performance improvements while maintaining power density at a status quo.

Unfortunately, the physical scaling of a transistor was not followed by the scaling of its electrical properties. A transistor conducts when the supply voltage is higher than a certain threshold voltage. However, the continued miniaturization of the transistor led to a substantial increase in leakage current. As a result, scaling the threshold voltage down proportional to the transistor size became impossible. This meant that the active and passive power consumption of a transistor did not decrease with scaling technology anymore. Therefore, putting an increasing number of smaller transistors on a chip kept the area constant, but not its power consumption, increasing power density.

The increase in power density generated an excess of heat which has to be dissipated. The necessity to dissipate this excess heat grew to be a major concern for processor architects. Multiple solutions were devised, with the birth of multi-core processors where two or more cores are placed on one chip being crucial. However, these multi-core processors still suffer from excess heat generation, albeit less severe, if the processor's clock frequency is kept reasonably low. One promising direction to mitigate excess heat generation is to improve the energy efficiency of the processors through specialization for (a) specific application(s). This type of specialized processors is called Application Specific Instruction Processors (ASIP).

Application-specific cores are tailored to (a) specific application(s) by removing or reducing all components that are not used, or under-utilized, by the application(s) (e.g., smaller caches or a narrower pipeline), and/or enlarging and adding components that benefit the application (e.g., accelerators). Embedded processors are a typical use case for application-specific processors, because they execute a limited set of applications and can be tightly optimized. Moreover, processors operating in mobile devices are continuously power and/or energy constrained while still aiming to achieve the best performance. Thus, both types of processors are prime candidates for application-specific optimizations.

Unfortunately, contemporary superscalar out-of-order processors are incredibly complex to analyze and optimize. Much of this optimization work is performed using detailed simulation, which is very slow and can thus inhibit the full design space exploration needed for finding the specific optimizations. The main motivation for this thesis is to help processor architects with alternative tools that enable the development of application-specific processors to improve performance, power and energy efficiency.

1.2 Key Contributions

If designing application-specific processors requires fast design space exploration tools to optimize for a targeted application(s), an alternative to simulation is required. Analytical models are an excellent fit for such design space exploration as they provide fast performance predictions and insight into the interaction between an application's characteristics and the micro-architecture of a processor. The key contributions of this thesis all relate to improving a pre-existing analytical model helping to prune large design spaces.

1.2.1 Micro-architectural Independent Analytical Model

Current analytical models require some micro-architecture dependent inputs, such as cache miss rates, branch misprediction rates and memory-level parallelism. This requires profiling the applications for each cache, branch predictor and reorder buffer (ROB) configuration of interest, which is significantly time-consuming compared to evaluating the actual analytical models. In this work we present a *micro-architecture independent* profiler and associated analytical models that allow us to produce performance *and* power estimates, based on a single profiling run, across a large design space almost instantaneously.

We show that using a micro-architecture independent profile leads to a speedup of 300× compared to detailed simulation for our evaluated design space. Over a large design space, the proposed model has an average error of 13% for performance and 7% error for power predictions, compared to cycle-level simulation. The model is able to accurately determine the optimal processor configuration for different applications under power or performance constraints, and provides insight into performance through cycle stacks.

This work was published at: S. Van den Steen, S. De Pestel, M. Mechri, S. Eyerman, T. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. Micro-architecure independent analytical processor performance and power modeling. In *Proceedings of the International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 32–41, 2015

This paper was nominated as one of the candidates for the best paper award at the 2015 ISPASS conference.

1.2.2 Sampled Model Evaluation

The previously proposed model sampled the workload during profiling to limit profiling time, after which the profiles for the different samples were combined to create an average profile. This profile served as an input to the analytical model. The key insight in this work is that we can improve the prediction accuracy by evaluating the model for all samples separately and combining the performance predictions across the samples. For contention modeling this makes sense as it is important to look at individual samples to predict bursty behavior rather than averaged out samples. Furthermore, some statistics such as cache misses and Memory-Level Parallelism (MLP) can influence each other but by averaging out both, the model misses certain behaviors that cancel out or reinforce each other.

Over a large design space, the improved model has a 9.3% average error for performance and a 4.3% average error for power, compared to detailed cycle-level simulation. Besides offering insight in the performance losses of an application running on a specific processor through cycles stacks, we employ the model to build Pareto plots. These Pareto plots offer insight into the performance-power trade-offs when picking either low-power or high-performance processors.

This work was published at: S. Van den Steen, S. Eyerman, S. De Pestel, M. Mechri, T. E. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. Analytical processor performance and power modeling using micro-architecture independent characteristics. *IEEE Transactions on Computers (TC)*, 65(12):3537–3551, 2016

This paper was chosen as the featured paper of December 2016 and a video[1] explaining the work was uploaded to the IEEE Transactions on Computers YouTube channel.

[1]`https://youtu.be/g3cDPM54YFA`

1.2.3 Modeling Memory-Level Parallelism

The previously proposed analytical models relied heavily on cold cache misses to predict MLP. This technique works well for evaluating relatively short traces without the need for a significant warmup. However, it falls short when evaluating complete benchmarks. Hence, we develop a new technique to model MLP that relies on the stride behavior of static load instructions. We profile the workload once and measure a set of distributions to characterize the workload's inherent memory behavior. We subsequently generate a virtual instruction stream, over which we then process an abstract MLP model to predict MLP for a particular micro-architecture with a given reorder buffer and last-level cache (LLC) size. We also take the impact of the miss status handling registers (MSHR) and a stride-based prefetcher into account. Experimental evaluation reports an improvement in modeling error from 16.9% for the cold-miss MLP model to 3.6% on average for the stride-based MLP model for predicting the average time the processor has to wait on DRAM in the presence of a stride-prefetcher.

This work was published at: S. Van den Steen and L. Eeckhout. Modeling superscalar processor memory-level parallelism. *IEEE Computer Architecture Letters (CAL)*, 17(1):9–12, 2018

1.2.4 Open Sourced Framework

The framework developed during this thesis is publicly available on GitHub. It consists of the Architecture Independent Profiler (AIP) available at `https://github.com/samvandensteen/AIP`, and the Processor Modeling Tool, available at `https://github.com/samvandensteen/PMT`. The tools are licensed under a GNU GPLv3 license.

To execute the tools, a number of other libraries are needed. The minimum requirements to execute the tools on an x86-based system are the following:

- Pin 2.x
- Google Protobuf 3.x
- Python 2.x

Some features of the framework to speed up the profiling or automatically produce performance plots rely on having other packages installed such as:

- google-sparsehash
- python-matplotlib

1.3 Other Research Activities

During the course of my PhD, from January to July 2017, I had the opportunity to intern at ARM in Cambridge, UK. The goal of this internship was to modify the developed models to work within the ARM simulation infrastructure. This required some modifications to the work flow presented in my thesis. First, I implemented the Pin-based profiling tool [48] into the ARM Fast Models framework [1]. Second, I modified the analysis tool to model an ARM core more faithfully. This primarily meant enforcing stricter constraints on the obtainable performance. I compared the predicted processor performance results both to an internal cycle-accurate simulator and real hardware for the SPEC CPU 2000 [9], SPEC CPU 2006 [10] and numerous EEMBC benchmark suites [3].

The final results of this internship approached the average accuracy of the proposed x86-based analytical model. Unfortunately, there were some outliers which I was unable to fix due to the brevity of the internship. The reason for these outliers are likely caused by ARM-processors being more resource-constrained, both in the processor front-end and back-end, which proved to be more challenging to capture in the analytical model than anticipated. However, I strongly believe that there are no fundamental limitations as to why the analytical model cannot be extended and fine-tuned to more accurately model ARM processors.

1.4 Thesis Overview

The remainder of the thesis discusses the research and framework we developed building on the previously proposed mechanistic interval model [32] to enable fast design space exploration. The thesis is organized in the following chapters.

Chapter 2 introduces the necessary background to understand the thesis. We discuss how a superscalar out-of-order processor core is built. We explain the different pipeline stages and how an instruction is processed. Next, we clarify the difference between simulation, sampled simulation and both empirical and mechanistic modeling to obtain performance and power predictions. In the next section, we discuss how processor power consumption can be modeled using different tools. Afterwards, we introduce the interval model and describe how it predicts processor performance. Lastly, we explain the advantage of using a micro-architectural independent interval model.

In Chapter 3, we discuss the complications an x86-based processor introduces for predicting performance using the interval model. We explain the different modifications introduced to improve its accuracy. Furthermore, we show how branch misprediction rates can be calculated without simulating the branch predictor [22] and how we predict power consumption using activity factors.

Chapter 4 introduces the statistics required to predict the influence of memory accesses on our performance predictions. We explain how we can predict cache miss rates using a statistical model called StatStack [28] and discuss two different techniques to estimate MLP. Accurate predictions of the latter are crucial to arrive at accurate performance predictions. Furthermore, we discuss a number of constraints imposed to the parallel execution of memory requests by the MSHRs and memory bus. We also, briefly, show the importance of including a prefetcher model and how power consumption for the memory hierarchy can be predicted.

Chapter 5 discusses the sampling approaches we implemented to speed up the collection of statistics necessary to estimate performance and power for the micro-architectural independent interval model. It also discusses the errors introduced by these sampling techniques.

Chapters 6 and 7 show the results obtained with the mechanistic model in this thesis. First, we describe our experimental setup. Afterwards, we discuss the accuracy of our performance and power predictions for our reference architecture and a design space of processor configurations. We show that we can generate CPI stacks offering insight into the performance losses of an application executing on a specific processor architecture. Furthermore, we demonstrate that our model can track the phases of an application relatively accurately and that modeling a prefetcher (when present) is necessary. When applying this model, we introduce Pareto plots to enable design space exploration and choosing an optimal core for a given application. We include a brief discussion using machine learning techniques for exploring a design space. We show that they produce accurate performance and power predictions on average, but fail to accurately predict performance and power trends. Lastly, we show that our model can even be used to find the optimal design point when Dynamic Voltage and Frequency Scaling (DVFS) is used.

Finally, in Chapter 8 we conclude the thesis and discuss some possible future work to extend the model to new types of processor cores and applications.

Chapter 2

Background

2.1 Out-of-order Processor

This thesis focuses on modeling the performance of pipelined superscalar out-of-order processors[1]. To understand how to model processor performance and power, one needs to understand the architecture of a processor. First and foremost, it is important to understand the different terms defining this type of processor. Pipelined execution means that processing an instruction takes multiple time steps (cycles) and different instructions reside in different stages of the processor pipeline in a given cycle. This is referred to parallelism in time. Superscalar execution means there are multiple, parallel pipelines through the complete processor. Hence, multiple instructions can enter the processor's parallel pipelines at the same time step which is called parallelism in space. Out-of-order execution means that the processor supports execution of instructions out of program order to improve performance, but the software is given the illusion of execution instructions in the order as specified by the programmer. The combination of both concepts leads to pipelined superscalar out-of-order processors, which offer high-performance, but unfortunately also consume a lot of power.

When an instruction is executed by the processor, it is flowing through a pipeline consisting of multiple stages. These stages are often again divided in multiple steps to exploit as much parallelism as possible. The stages of a processor are Instruction Fetch (IF), Instruction Decode (ID), Operand Fetch (OF), Execution (EX), Memory Access (MEM) and Write-Back (WB). The IF, ID and OF stages form the processor front-end, while the EX, MEM and WB stage comprise the back-end. Note that not every instruction needs all stages. Some instructions do not access any memory and others, e.g., unconditional branches, do not need any operands. Figure 2.1 shows a schematic overview of an out-of-order processor.

[1]From now on, when we use the term processor, it denotes a pipelined superscalar out-of-order processor, unless mentioned otherwise.

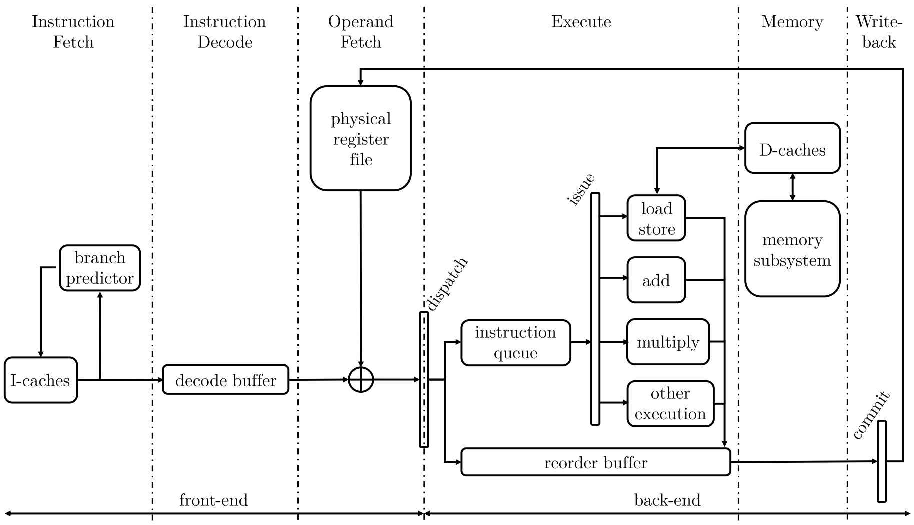


Figure 2.1: Schematic overview of the pipeline of an out-of-order processor.

The Instruction Fetch stage of the processor is responsible for fetching the next instruction(s). When an instruction is fetched that changes the flow of the program (e.g., it jumps to another part of the program), it is unclear what the next instruction is. Therefore the IF stage of a processor also features a branch predictor, which predicts the next instruction. This allows for the IF stage to keep fetching correct-path instructions with a minimal delay provided the branch predictor performs well.

The Instruction Decode stage translates the bytes of an encoded instruction to a format that the processor understands. It determines the type of the instructions and figures out their operands and registers. Furthermore, if an instruction performs a complex operation, the ID stage can split it up into smaller parts, called micro-operations. All of this information is sent to the next stages in the processor pipeline.

```
1   $R_1 \leftarrow R_2 + R_3$
2   .....
3   $R_5 \leftarrow R_1 + R_4$
4   .....
5   $R_1 \leftarrow R_6 \times R_7$
```

Example 2.1: Illustration of the different types of register dependences.

In the ID stage, the processor will also rename the operand registers to eliminate false dependences. We distinguish between three types of dependences, namely real dependences or Read-after-Write (RAW) dependences, anti-dependences or Write-after-Read (WAR) dependences, and output dependences or Write-after-Write (WAW) dependences. The latter two are false dependences that can be eliminated through register renaming. Example 2.1 illustrates why the renaming step is necessary in out-of-order execution. Instruction 1 calculates the addition of R_2 with R_3 and stores the result in R_1 which is consumed by instruction 3, thus there exists a RAW dependence between instructions 1 and 3 through register R_1. This type of dependence cannot be eliminated by register renaming. Instructions 1 and 5 exhibit a WAW dependence because they both write to register R_1. Instructions 3 and 5 are an example of a WAR dependence since instruction 3 reads R_1 and instruction 5 overwrites R_1. For a superscalar out-of-order processor to work efficiently and correctly, both false dependences have to be eliminated. After all, instruction 5 does not depend on any previous instruction and, to maximize performance, an out-of-order processor can execute it before instruction 3. However, R_1 now contains a newer value which is not the correct input for instruction 3. The processor still needs to supply the older, correct result to instruction 3. Example 2.2 illustrates the solution to this problem by renaming registers. Instruction 1 writes to a different physical register F_1, which is consumed by instruction 3, and instruction 5 writes to register F_2. The processor can now execute these instructions in any order and still arrive at the correct result. Note that in this example only the registers through which there are dependences are renamed, but in reality all registers will be renamed.

```
1   $F_1 \leftarrow R_2 + R_3$
2   .....
3   $R_5 \leftarrow F_1 + R_4$
4   .....
5   $F_2 \leftarrow R_6 \times R_7$
```

Example 2.2: Code example with renamed registers.

In the Operand Fetch stage, the register operands that are needed to execute an instruction are gathered. Here, the processor will read the physical register file. Note that, at that point in the pipeline, not every instruction has all of its operands available immediately. Example 2.2 already shows that instruction 3 depends on the result of instruction 1. Even with the missing operand(s), instruction 3 will still progress to the next pipeline stage.

After the OF stage, the instruction will be dispatched. This is where we enter the EX stage. In this stage the instruction is inserted into both the instruction queue and reorder buffer. Afterwards, if all operands are available, the instruction is issued to a so-called functional unit for execution. Depending on the type of instruction, it will be sent to a different functional unit (e.g., an addition or multiplier unit). If the instruction depends on the result of a previous instruction and has not received all of its operands in the OF-stage, it will wait in the instruction queue until that operand can be forwarded from one of the functional units. The EX-stage is the stage from which out-of-order execution derives its name. The processor will traverse the instruction queue and if an instruction has all its operands available, it will issue and execute the instruction. Because the processor can traverse the entire instruction queue and pick any instruction, instructions are not per se executed in program order.

If an instruction has to access memory, it will enter the Memory stage. In the previous stage, the functional unit will have calculated the address where to read or write the data in memory. It will first query the cache hierarchy which serves as intermediary fast storage, and if the data is not found there, request it from the memory subsystem. Depending on which level of cache or main memory the data has to be fetched from, this can take a long time.

The last stage is the WB stage. Here the results are committed to the physical register file and/or the memory hierarchy. The instructions are removed from the ROB. This operation is executed in-order, meaning that the instructions are taken out of the ROB in the same order they were put in. This guarantees that, despite internal reordering of instructions, the architectural state of the processor is always correct and consistent, which is particularly important in the context of precise exceptions.

2.2 Architectural Simulation

Physically producing a processor is so expensive and time consuming that it is impossible to evaluate improvements to the design of a processor by actually prototyping it. Instead, processor architects often rely on simulation to predict the performance of a (set of) program(s) executing on a processor. Generally these different simulation techniques can be classified according to the level of abstraction the processor designer employs. One could use very detailed simulation of every part of the processor, but it is also possible to omit or abstract parts of the processor. Truthfully simulating every part of a processor produces very accurate performance predictions, but is also extremely slow. Therefore, if fast results are required, the trade-off can be made to omit or abstract away as much of the processor as possible.

2.2.1 Timing Simulation versus Functional Simulation

Many types of simulations exist, each with their own trade-offs with respect to speed and prediction accuracy. We briefly discuss the most important categories and highlight the difference between timing simulation versus functional simulation.

Two types of timing-focused simulation are cycle-accurate simulation (or RTL-simulation) and cycle-level simulation. A processor architect will employ a timing-based simulation to figure out how a processor executes a program, but also when it performs specific actions.

Cycle-accurate simulation will model every structure of the processor in software. Cycle-accurate simulators are able to simulate what happens inside the processor on a cycle-by-cycle basis. Hence, the produced performance predictions are usually accurate within a couple of percent compared to real hardware.

However, cycle-accurate simulators are typically very slow. They reach simulation speeds of up to 1 kHz, meaning that they are able to simulate 1000 processor cycles per second in real time. This is six orders of magnitude slower than a real system. Thus, if executing a program on a real system would take 1 s, simulating it using cycle-accurate simulation would take roughly 10 days. It is obvious why it is infeasible to generate performance predictions for multiple applications and processor designs.

Furthermore, most cycle-accurate simulators are not available to the academic world. The reason is that if a processor company would release its simulator, they would risk a third-party being able to find out how that processor works exactly. Since developing a processor is extremely expensive, it is understandable that companies do not release their simulators.

Luckily, there are a number of simulators available to academics to perform cycle-level simulation. These simulators still employ a timing model, but they tend to simplify the simulation of certain parts of the processor. Examples include SimpleScalar [16], Gem5 [14], PTLSim [74], Graphite [50], Sniper [17], etc. They are usually one or two orders of magnitude faster than true cycle-accurate simulators. Furthermore, the performance predictions are still relatively accurate and the simulators are usually validated against real hardware.

Functional simulation abstracts away the timing-related details and is thus not used to predict processor performance. These simulators focus on what happens in the processor, rather than when it happens. As a result, they are one or two orders of magnitude faster than cycle-level simulation. Many of the previously cited simulators (e.g., SimpleScalar [16] and Gem5 [14]) also offer a functional simulation model.

Functional simulation can also focus on only simulating specific parts of the processor, e.g., the cache hierarchy or branch predictor. Some simulators, e.g., Sniper [17], offer a cache-only simulation mode. This can be useful if the focus is to optimize one specific structure in the processor. Furthermore, the results of a cache or branch predictor simulator can be used as input to analytical models such as the interval model, see Section 2.5.

A tool that is widely used to build (functional) simulators is Pin [48], a binary instrumentation framework developed by Intel for x86-applications. This tool allows to instrument an application as it is executed, collecting statistics from the dynamic instruction stream. The collected statistics can be used to build a full timing-based simulator such as Graphite [50] and Sniper [17] as well as, for example, a functional cache simulator for multi-core processors such as CMP$im [39].

2.2.2 Sampled Simulation

Speeding up simulation can often be achieved through sampling. Sampled simulation has been researched extensively since it introduces two major challenges, namely the selection of a representative (set of) sample(s) and starting from a 'correct' (micro-)architectural state at each sample.

The selection of a representative (set of) sample(s) to simulate instead of simulating a complete application can be solved in one of two ways. Either a statistical approach is taken where samples are taken in a random fashion or periodically, or a targeted approach is used where the program is first analyzed to find representative samples.

Random sampling to simulate processor performance was first employed by Conte et al. [21]. The idea behind this random sampling approach is that it allowed to collect an unbiased, representative set of samples. Wunderlich et al. [73] proposed SMARTS which employs systematic or periodic sampling instead. Periodic sampling can produce unrepresentative samples if a program exhibits periodic behavior. However, few applications exhibit the same level of periodic behavior throughout their complete execution, which results in this being an unlikely problem. The main advantage of both statistical sampling methods is that, through the central limit theorem [47], one can prove that, for any performance metric, the sampled mean will approach the true mean within a some confidence interval depending on the number of samples.

Using the targeted sampling approach requires to preprocess a program after which a (set of) representative sample(s) is chosen and associated with a weight. The most straightforward way is to select a sample based on, for example, functional cache and branch simulations, which was demonstrated by Skadron et al. [61]. However, the disadvantage is that the sample is not necessarily representative for other micro-architectures. Therefore, it is necessary to select micro-architecture independent metrics and analyze those to find representative samples. The most well-known approach is SimPoint [59]. This work divides a program in intervals, builds Basic Block Vectors [58] for them by counting how many times basic blocks are executed, and clusters those to find similar intervals. SimPoint assumes that intervals with similar basic block behavior will exhibit similar micro-architecture behavior and thus only one interval from a cluster needs to be simulated. The SimPoint approach is employed by Patil et al. [54] to enable deterministic sample replay using Pin [48].

At the beginning of execution-driven simulations of (a) representative sample(s), it is absolutely necessary for the processor's architectural state to be correct. This entails that the processor's registers and the memory content exactly match the content as if the application was simulated completely. Otherwise, the behavior of that sample may be different from the real behavior prompting incorrect conclusions. Starting from the correct architectural state can be achieved either through fast-forwarding or checkpointing. The former can be achieved using functional simulation, which can be slow for long-running programs, or using execution on real hardware as established by Szwed et al. [62].

The latter will dump the complete architectural state to a checkpoint file. These files can be very large as shown by Van Biesbrouck et al. [66], but this can be optimized by, e.g., only saving the memory that is used in the sample [66, 72].

Next to the architectural state being correct, it is also advisable for the micro-architectural state to be as accurate as possible. This involves an accurate cache, translation look-aside buffers (TLB), branch predictor and processor core state. This is necessary for the sample to be truly representative. Otherwise, a sample might exhibit, e.g., different memory behavior because it generates more cache misses. A lot of work focuses on achieving an accurate cache state either through prepending a warm up phase to the sample [26, 36], leveraging reuse distances to estimate a cache state [52], or checkpointing the micro-architectural state [71]. The downside of the last solution is that it is dependent on one specific micro-architecture, while the former is not. Similarly, warming up the branch predictor can also be achieved by prepending a warmup phase [21]. An accurate processor state is of lesser importance because sample units often are millions of instructions long while there are only a couple of hundreds of instructions in-flight at a time in the processor. Thus, after simulating the first couple of hundreds instructions, which have little influence on the total sample execution, the processor state is warmed up automatically. However, if short samples are used, it may be necessary to prepend a warmup phase to estimate the processor core state as demonstrated in SMARTS [73].

2.3 Performance Modeling

An orthogonal approach to (sampled) simulation is to use mathematical models that model the interactions in a processor. The models themselves do not simulate any part of the processor, although they often require inputs obtained through simulation. If one ignores the required input generation, evaluation of these models is very fast. Of course, simulation and modeling can also be combined where first a model is used to search for interesting designs and then rely on simulation to obtain more accurate predictions in a region of interest.

2.3.1 Empirical Modeling

Empirical models are one subset of models which try to predict a (set of) metric(s) based on a training set of results for similar experiments. These models are based on the premise that current processor micro-architectures are too complex to model, but that through machine learning one can calibrate a generic model to faithfully reproduce their behavior. They are often called black-box models because they take a set of inputs and produce a result without the user knowing why it produces that result. Generating a prediction for a program running on a processor requires two steps. First, a prediction model is built by training the mathematical model using the results of a training set. For this, a number of inputs and outputs from a set of simulation experiments have to be provided. The inputs will, for example, quantify the size of certain structures in the processor and include characteristics of an application running on that processor. The supplied outputs need to be the metrics which the model needs to predict, e.g., performance or power. The second step is the evaluation of the model for a different processor architecture. By providing the same set of processor and/or program inputs it can predict the (set of) metric(s) for which it was trained. The evaluation of such a model is usually fast.

However, there are a couple of important downsides to this technique. First of all, because it relies on a (large) set of simulations, building the model is inherently slow. The reason for this is that a set of simulation results needs to be collected as input for the training of the model. Second, this kind of model offers relatively little insight as it only produces the metric(s) for which it was trained without divulging the reason for this prediction (a black-box approach). Third, because of the training step, overfitting the model is an important risk. An overfitted model produces good predictions if the inputs are similar to the ones on which it was trained. But, due to the overfitted nature, it may produce worse results when predicting a metric for an input outside of the training scope.

Building an empirical model is relatively easy and is thus often a useful technique. Lee et al. [43] and Ipek et al. [38] built accurate performance prediction models using linear regression and artificial neural networks, respectively. A significant body of work extended on these empirical models. Lee et al. [44, 45] further improve their previously proposed models and show design space exploration applications. Azizi et al. [11] include power and energy metrics to find power-performance trade-offs. Singh et al. [60] explore the possibility of using performance counters on real hardware to perform real-time power modeling and scheduling.

2.3.2 Mechanistic Modeling

Another approach for predicting performance is through mechanistic analytical modeling. For this technique a set of equations are built that try to capture the inner workings of a processor. A mechanistic analytical model builds on simplifying assumptions and first-order effects, observed by studying the flow of instructions through the processor pipeline. It tries to capture the interactions between the hardware (the processor and memory), and the software executing on it (one or more programs).

Over the years, a significant amount of research was performed on how to study and model processor performance. Emma et al. [29] show how CPI stacks can help understand performance bottlenecks. Michaud et al. [49] quantify the influence of the instruction fetch bandwidth on performance with respect to branch mispredictions and instruction level parallelism (ILP). Hartstein et al. [35] introduce a model that details how the optimal pipeline length can change as function of the ILP and pipeline stalls. A first-order model focusing on pipeline stalls due to miss events was developed by Karkhanis et al. [42].

This work models processor performance at the issue stage. Eyerman et al. [32] further elaborated on the latter work and introduced the interval model which models performance at the dispatch stage. Jongerius et al. [40, 41] proposed an ISA-independent mechanistic model using the LLVM [7] intermediary format to predict processor performance.

Mechanistic models are generally less accurate than empirical models, and do not model the whole processor in detail. However, while mechanistic models have limited detail, they do reveal how the program interacts with the micro-architecture through the mathematical equations upon which they are built. This also offers the benefit of easy modifiability and extensibility since every step to the end result can be traced back to a part of the equations.

Mechanistic modeling also consists of two phases: application profiling and performance estimation. Profiling is usually the most time-consuming step, as the instructions of the application need to be analyzed to obtain the application characteristics required by the mathematical model. However, using proper sampling techniques, it can be sped up by several orders of magnitude compared to simulation. The performance estimation step is significantly faster since it solely consists of evaluating a set of equations. This step can be completed in a couple minutes, depending on the 'size' of the prediction problem.

2.4 Power Modeling

Processor power consumption can be split into two components, static and dynamic power consumption. Static power consumption is related to the transistor technology and is a function of the leakage current, I_l, and the supply voltage, V_{dd}. Note that the leakage current, I_l, also depends on temperature, indirectly causing static power consumption to also depend on temperature.

$$P_s = I_l \times V_{dd} \tag{2.1}$$

The dynamic power consumption, shown in Equation 2.2, is the power consumed due to transistor switching and depends on a number of factors including the capacitive load, C, the supply voltage, V_{dd}, the processor frequency, f, and the activity factor or transistor switching activity, a.

$$P_d = \frac{1}{2} \times C \times V_{dd}^2 \times a \times f \tag{2.2}$$

The relative contribution of both types of power consumption is difficult to generalize. Traditionally, dynamic power consumption was more important than static power consumption, but with new technologies, static power consumption becomes increasingly more important. As an example, in our experiments, for a traditional 45nm CMOS processor, the static power consumption is around 40% of the total power consumption.

The introduction already indicated that power management is one of the most important considerations when developing processor chips. Hence, different techniques to model power consumption have been developed. Similar to RTL-based performance simulations, power simulations can be very detailed, simulating power usage at the transistor level. However, these simulators are also very slow and not available to academia because they reveal too much of the processor architecture.

Therefore, tools like CACTI [63], Wattch [15] and McPAT [46] were developed that model power consumption at higher abstraction levels. CACTI was developed to model power consumption in the memory hierarchy, but is also commonly used to obtain power estimations of other SRAM[2] structures. CACTI uses data provided by the International Technology Roadmap for Semiconductors (ITRS) [6] to estimate transistor-level power numbers. Besides power consumption, it also models the area and timing constraints of these structures. Wattch [15] abstracts this further, but uses the capacitance models from CACTI. It employs parameterized power models for different hardware structures and does not simulate area or timing constraints. To estimate power it relies on activity factors for the hardware structures. For example, the number of additions in a program is counted through architectural cycle-level timing simulation, which can then be used to calculate the power usage of a parameterized addition functional unit. McPAT [46] integrates both area, timing and power in one tool. For both area and timing constraints, it builds on the models proposed in CACTI. Similar to Wattch, power is predicted using activity factors. McPAT uses an XML interface to describe the hardware structures and to summarize all accesses to these processor structures. These tools can produce power predictions accurately within 20% of the actual power consumption.

An extra complication the above tools omit is that both static and dynamic power consumption is also influenced by the temperature of the processor and vice versa. HotSpot [37] can be used to develop a compact thermal model for a processor showing both static and transient temperature information. This thermal model can then be used as feedback to study how temperature influences power usage.

2.5 Interval Model

In this thesis we focus on modeling a processor using the mechanistic approach. Because contemporary processors are complex, predicting the performance of a processor is also a complex task. We focus on extending the interval model [32]. This model has as upsides that it is relatively simple to understand, is still relatively accurate and offers a lot of insight into processor/program performance.

Contrary to the initial interval-based models [42, 49], which focused on the fetch or issue stage, the interval model [32] predicts processor performance from the viewpoint of the dispatch stage. There are two main reasons for this approach. The first reason is that at this point in the processor both front-end

[2]SRAM: Static Random Access Memory

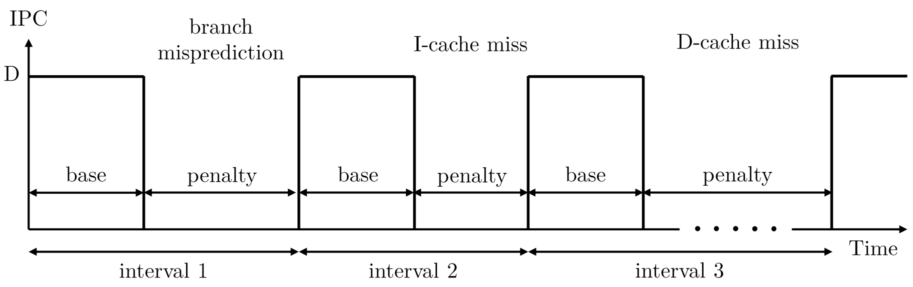


Figure 2.2: Program execution split up into separate intervals following the interval model.

and back-end issues can be analyzed. At other stages in the processor, the miss events causing the issue can be hidden under other events which limits insight into the processor's performance. Secondly, at the dispatch stage, a clear on-off behavior can be observed: either instructions are being dispatched or a structural problem arose that inhibits dispatching instructions.

The interval model is built upon the observation that miss events can stall the processor pipeline which leads to a degradation in average processor performance. Figure 2.2 shows the categories of miss events that can deteriorate performance. The interval model distinguishes between front-end misses, classifying them as either branch mispredictions or I-cache misses, and back-end misses or long-latency load D-cache misses. Important to notice is that, in an out-of-order processor, different D-cache misses can overlap and this needs to be modeled as accurately as possible.

Under normal operation, the interval model operates under the assumption that the processor can achieve a performance or instructions per cycle (IPC) equal to the processor's dispatch width D as indicated in Figure 2.2. However, when the processor encounters one of the previously defined miss events, (IPC) drops to 0. Depending on the miss event, the time to resolve it can range from a couple of cycles to a couple of hundreds of cycles. The respective miss events are collected using functional simulation of the cache hierarchy, branch predictor and ROB. Note that this type of simulation is faster than full-blown timing simulation of the complete processor, but it is still time-consuming. The goal of the interval model is to predict the time it takes to resolve all miss events and thus arrive at an accurate performance prediction.

2.5.1 I-Cache Misses

In order to make forward progress, a processor continuously fetches new instructions. These instructions are stored either in the cache or in main memory. Depending on the level of the memory hierarchy the processor needs to access, this operation takes one to hundreds of cycles. If the processor finds the instruction in the first level of the instruction cache, it is always able to progress. If it has to fetch the instruction from a lower level, the processor will

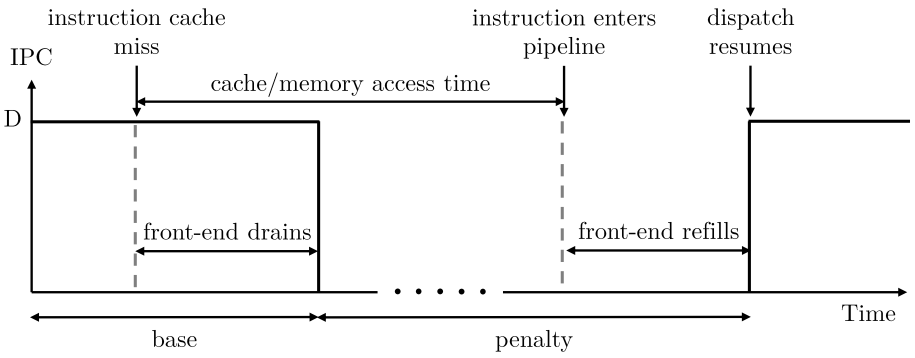


Figure 2.3: Interval breakdown of an instruction cache miss.

have to wait multiple cycles before it can supply a new instruction to execute. Figure 2.3 breaks down the different steps of an instruction cache miss.

The processor front-end exists of multiple stages that still contain useful instructions when the I-cache miss occurs. These instructions can still progress through the pipeline, but no new instructions are fetched, and thus the front-end pipeline drains. The dispatch of instructions does not halt immediately, but only when the front-end pipeline is empty. The stall is resolved once the requested instruction enters the pipeline. The time this takes is dependent on the level of the memory hierarchy that supplies the instruction. At that moment the front-end starts to refill and when the front-end is full, dispatch (and instruction execution) resumes. The penalty of an instruction cache miss is thus equal to the latency of fetching the instruction because the front-end drain and front-end refill time cancel each other out.

2.5.2 Branch Mispredictions

Branches are a type of instructions that control the direction the program is taking. Depending on the direction of the branch, the program behavior will be different because the next instructions to be fetched and executed will be different.

```
variable a = ... load from memory ...
if a fulfills condition:
    ... perform if calculation ...
else:
    ... perform else calculation ...
```

Example 2.3: Code containing a branch instruction.

A pseudo-code example containing a branch instruction can be found in Example 2.3. Depending on the value of variable a, the next instructions are part of the calculations in the if-statement or else-statement. The value of variable

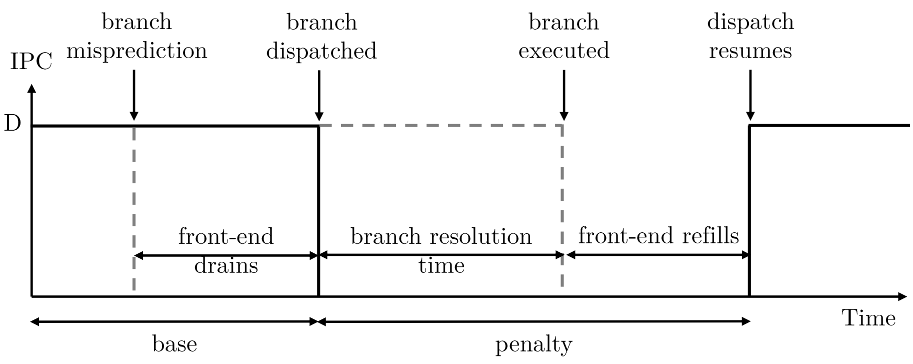


Figure 2.4: Interval breakdown of a branch misprediction.

a is not necessarily known when the branch instruction is fetched. Hence, in order to smoothly fetch the next instructions and not wait until variable a is known, the processor predicts the direction of the branch. The structure in the processor that performs this action is known as the branch predictor. Most branches are easy to predict because they are executed many times and their subsequent executions are correlated. However, for other branches the branch predictor might predict the direction incorrectly. This event is called a branch misprediction and can entail a significant penalty.

Figure 2.4 shows what happens when the branch predictor incorrectly predicts the branch direction. Unlike an I-cache miss, new instructions enter the pipeline. However, the new instructions are part of the wrong execution path and they will not contribute to the progress of the program. This is indicated by the dotted line. Hence, similar to the case of an instruction cache miss, the effect of a branch misprediction is a front-end drain step where previously fetched, correct-path instructions progress through the pipeline. When the front-end is drained of useful instructions, the branch is dispatched and useful program progress halts. Note that, while we visualize this as an IPC dropping to zero, the processor is actually still executing instructions. However, because the instructions that are dispatched are part of the wrong execution path, this is visualized as an effective IPC of zero. If the branch depends on the result of other instructions, it takes a number of cycles before the branch is executed. We call the time it takes from dispatch to execution of the branch the branch resolution time. After the branch is executed, the branch predictor knows whether its prediction was correct or incorrect. Because the processor now knows the correct execution path, it can start fetching correct-path instructions again, the front-end refills and when dispatch resumes, the program can progress. Thus, the penalty of a branch misprediction is the sum of the branch resolution time and the front-end refill time.

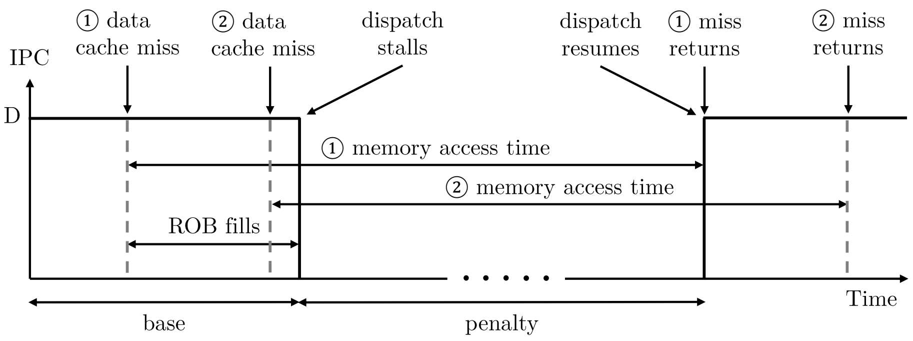


Figure 2.5: Interval breakdown of a long-latency load data cache miss.

2.5.3 D-Cache Misses

Executing instructions requires data on which to operate. This data, known as the data set of a program, will be loaded into main memory when required. Because main memory is a lot slower than the processor (about two orders of magnitude), processor architects add caches between main memory and the processor. These caches are organized in a hierarchy, where the top level is fast enough to deliver a continuous stream of data to the processor and the bottom level is about one order of magnitude slower. In the interval model it is assumed that the latency of accessing any of these intermediary cache levels can be hidden under the execution of other useful work. This is one of the key features of an out-of-order superscalar processor and is modeled as such. Hence, there is no observable performance penalty. However, the different cache levels have limited capacity ranging from a couple kilobytes to a couple megabytes. This capacity is often not enough to store the program's complete data set. When the data is not present in the cache, the processor has to access main memory, which takes up to hundreds of cycles. This long access latency causes the back-end structures of the processor to fill up and thus dispatch to stall. We call this event a long-latency D-cache miss.

Figure 2.5 shows the breakdown of the executing of a load instruction accessing main memory by the processor. When the data cache miss occurs, the missing instruction will advance to the head of the processor's ROB. Due to the long access time to main memory, the instruction will hit the head of the ROB before its data is returned. Since instructions need to leave the ROB in program order and the ROB is now full, this causes dispatch to stall. Before the instruction that generates the D-cache miss hits the head of the ROB, there are still instructions executed and committed. Hence, the penalty for a load data cache miss starts when it hits the ROB head and the ROB is full.

Contrary to instruction cache misses and branch mispredictions, multiple data cache misses can overlap. This is indicated in the figure with the data cache misses labeled using a one and two, which happen shortly after each other. The processor has already stalled due to the first data cache miss reaching the head of the ROB. However, during this stall, the second data cache miss has also requested its data from main memory. Hence, the latency of the second data cache miss is (partially) hidden under the latency of the first. This property of an out-of-order processor is called Memory-Level Parallelism (MLP). When the data for the first data cache miss returns, program execution can resume. Depending on when the second data cache miss happened, its data could be returned before it hits the head of the ROB. If this is the case, there will be no penalty for the second data cache miss as shown in Figure 2.5. It is however also possible that the latency cannot be hidden completely under the access latency of the first data cache miss. In the interval model, the simplifying assumption is made that the latency for any subsequent data cache miss that fits in the same ROB can be hidden completely. Note that the parallel processing of data cache misses does depend on the presence of non-blocking caches and MSHRs to keep track of outstanding load misses in the processor.

2.5.4 Interval Model Equation

Taking the above defined miss events into account to predict performance leads to Equation 2.3. The result of this equation, written as C in the equation's left-hand side, is the number of cycles it takes to execute a program.

$$C = \frac{N}{D} + m_{bpred} \times (c_{res} + c_{fe}) + \sum_i m_{ILi} \times c_{Li+1} + \frac{m_{LLC} \times c_{mem}}{MLP} \tag{2.3}$$

The first term in the equation is the number of instructions N divided by the dispatch rate, equal to the dispatch width D. This is the program's base performance and equals the maximum achievable performance. Thus, it takes at least $\frac{N}{D}$ cycles to execute the complete program on a processor with a pipeline width equal to D. The next three terms are penalties related to the respective miss events listed above. The second term consists of the number of branch mispredictions m_{bpred} multiplied by its penalty. As discussed in Section 2.5.2, this penalty is the sum of the branch resolution time c_{res} and the front-end refill time c_{fe}. Note that c_{fe} is a fixed constant only dependent on the micro-architecture. The next part of the equation is the sum of all instruction cache misses m_{ILi} per level multiplied by the access latency to the next level c_{Li+1}. The last term is equal to the number of last level cache misses (LLC misses), m_{LLC}, multiplied by the main memory access time c_{mem} and divided by the amount of memory-level parallelism MLP. Here, MLP is defined as the average number of outstanding long-latency load misses if at least one is outstanding [20]. This division models that, as discussed in Section 2.5.3, load accesses to main memory can be processed concurrently. Note that in the original model there is also a term for the dispatch inefficiency. However, due the extensions made to the model in this work with respect to the dispatch rate, including this term makes less sense and we do not discuss it here.

Some of the inputs in this equation are exclusively dependent on the program, some are dependent on the micro-architecture of the processor and others are dependent on a combination of both. The number of instructions, the

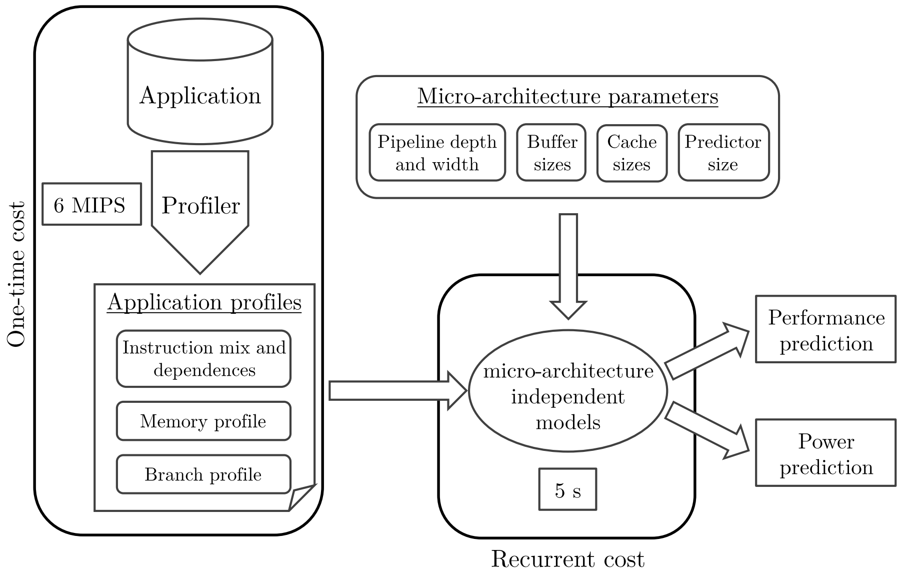


Figure 2.6: Schematic overview of the micro-architecture independent interval model to predict processor performance and power.

dependences between them and instruction mix are exclusively application dependent. On the other hand the dispatch width, the front-end refill time, access times to the different levels in the cache hierarchy or main memory, and the size of the ROB are features ingrained in the processor's micro-architecture. The other inputs, which are the number of branch and cache misses as well as the MLP are dependent on both the application and micro-architecture.

2.6 Micro-Architecture Independent Modeling

All of the previously discussed mechanistic models have one important downside: they use simulation-based inputs. As an example, the interval model requires inputs from a branch predictor simulator, a cache simulator and an MLP simulator. The consequence is that, even though the necessary simulations are not full-blown timing-based simulations, they are still costly. Especially the MLP simulation is time-consuming since it requires stepping over the complete instruction stream, instruction by instruction, to mark the dependences and correlate the load instructions with cache misses obtained from a cache simulator. The cost of the MLP simulation can be amortized by using an algorithm such as the one described in Eyerman et al. [30] which obtains MLP statistics for different sizes of the ROB in a single simulation run.

However, the problem of slow simulations is augmented by the fact that searching for energy-efficient processors designs requires exploring a large design space. To compare different processor designs from such a design space multiple simulations are needed. For example, it might be needed to explore

Instruction type	Frequency
Load	30%
Store	20%
ALU	15%
Multipy	10%
Branch	15%
Move	10%

Table 2.1: Example of an instruction mix of an application.

different cache hierarchies which requires different cache simulations which in turn impacts the MLP simulations. Obtaining the inputs to a mechanistic model using a significant amount of simulations is undesirable because of time constraints.

One way to mitigate this is to eliminate the simulation-based inputs by relying on application characteristics that are independent of the micro-architecture of the processor. This does not necessarily require modifying how a mechanistic model calculates the final performance, but requires including an intermediary step that transforms the micro-architecture independent application characteristics to miss events based on statistical models. One way of achieving this is described in more detail in Chapters 3 and 4.

Figure 2.6 shows a schematic overview of how a micro-architecture independent model predicts performance and power. Similar to any generic mechanistic model, a profiling phase is needed to collect profiles for (a set of) application(s). These profiles are usually a set of distributions that include information about e.g., the instruction mix and its dependences, memory behavior and branch behavior. Note that this profiling phase is a one-time cost. A given application needs to be profiled only once after which performance and power can be predicted for a complete design space.

Table 2.1 shows an example of one of the collected profiles, the instruction mix profile, where 50% of the instructions are accessing memory (loads and stores), 25% are compute instructions (ALU and multiply), 15% are meant to control the flow of the program (Branch) and the other 10% are generic instructions that move data around. This profile is necessary to predict the average instruction latency and possible contention in the processor's issue stage (see Sections 3.3 and 3.4).

The profiling phase is a one-time cost as the profiles contain only micro-architecture independent characteristics. At an indicative speed of six million of instructions per second (MIPS), it is also relatively fast compared to the previously required simulations. The statistics serve as input to a set of statistical models that combine them with a set of parameters describing the processor structures to transform them into, among others, miss events. The reason why this step is fast is because part of the cost of these statistical models can be amortized over multiple experiments. The mechanistic model can then calculate the performance and/or power based on the miss events and their associated latencies. Note that the original interval model does not calculate power consumption and that this is an extension. The complete work flow of collecting micro-architecture independent characteristics, predicting miss events and performance is significantly faster than both timing-based simulation and a mechanistic model using simulation inputs as detailed in Section 6.2.

Chapter 3

Modeling the Core

3.1 Improvements to the Interval Model

The interval model was originally developed for the Alpha ISA, which has been deprecated. In this work, the model is modified to work for modern x86-processors. The original interval model assumed a balanced processor meaning that every processor stage can sustain the same throughput of instructions as the previous stage. Contemporary processors are incredibly complex and are optimized to support many different types of workloads. If a specific workload stresses one part of the processor (e.g., the floating point functional units), this sustained throughput is not always achievable. Thus we introduce new penalty terms and modeling techniques to model constraints imposed by the imbalance between different processor stages. The following list summarizes the changes introduced over the interval model from Equation 2.3:

- The number of instructions N is replaced by a smaller unit of work called micro-operations.
- The base performance is now limited by the effective dispatch rate D_{eff} rather than the physical dispatch width D.
- The branch resolution time c_{res} is calculated using the average branch path.
- The latency to main memory also takes queuing delay over the memory bus into account.
- A new term called the LLC chain penalty, P_{hLLC}, is added.
- All miss events and the MLP (Memory-Level Parallelism) are calculated using statistical techniques, rather than a simulator.

All of the improvements discussed in this work lead to a new, micro-architecture independent performance and power model. Equation 3.1 summarizes how performance is calculated using the improved interval model. The following two chapters discuss the techniques to compute the required inputs without any type of micro-architecture specific simulation and the additional penalties to better model contention to improve the accuracy of the interval model.

$$C = \frac{N}{D_{eff}} + m_{bpred}(c_{res} + c_{fe}) + \sum_i m_{ILi}c_{Li+1} + \frac{m_{LLC}(c_{mem} + c_{bus})}{MLP} + P_{hLLC} \quad (3.1)$$

3.2 CISC versus RISC

One important characteristic to classify a processor is whether it uses a load/store architecture or a register-memory architecture. In a load/store architecture, there is a strict distinction between memory operations and Arithmetic Logic Unit (ALU) operations. Hence, arithmetic operations can only use registers and if new data is needed, a load has to fetch it from memory to a register. In a register-memory machine, ALU operations operate on both values from registers and memory.

Load/store architectures are used in Reduced Instruction Set Computer (RISC) processor designs, while register memory architectures use a Complex Instruction Set Computer (CISC). A CISC instruction-set architecture (ISA) can thus execute multiple, functionally different low-level operations as part of one instruction. Modern processors will often split instructions into different micro-operations in the decode stage such that it can more easily process them.

The original interval model was developed for the Alpha ISA, an older RISC architecture which used instructions as the smallest unit of work. This thesis focuses on the Intel x86 ISA, a CISC architecture. Equation 2.3 shows that to calculate the base performance, the number of instructions in a program is an input. Because the interval model calculates performance at the dispatch stage and x86 splits instructions into micro-operations before that stage, we need to take this into account. Therefore, we first compute the sequence of micro-operations from the x86 code. As a result, the N in Equation 3.1 is equal to the number of micro-operations, and not the number of instructions.

Figure 3.1 shows the ratio of the number of micro-operations to instructions for all benchmarks from the SPEC CPU 2006 suite. This breakdown into micro-operations was performed using the Intel X86 Encoder Decoder (XED) [5]. Note that this ratio varies a lot across benchmarks. For the *lbm* benchmark the ratio is close to 1.07, while for the *GemsFDTD* benchmark it is close to 1.38. This large difference indicates the necessity to take micro-operations into account in the model. After all, if we dispatch 1 billion instructions for both benchmarks, dispatching all micro-operations on a processor with a dispatch width of four will take approximately 267 million cycles for *lbm*, but it will take 345 million cycles to dispatch the same number of instructions for *GemsFDTD*. In the first case, the prediction error compared to using instructions would be

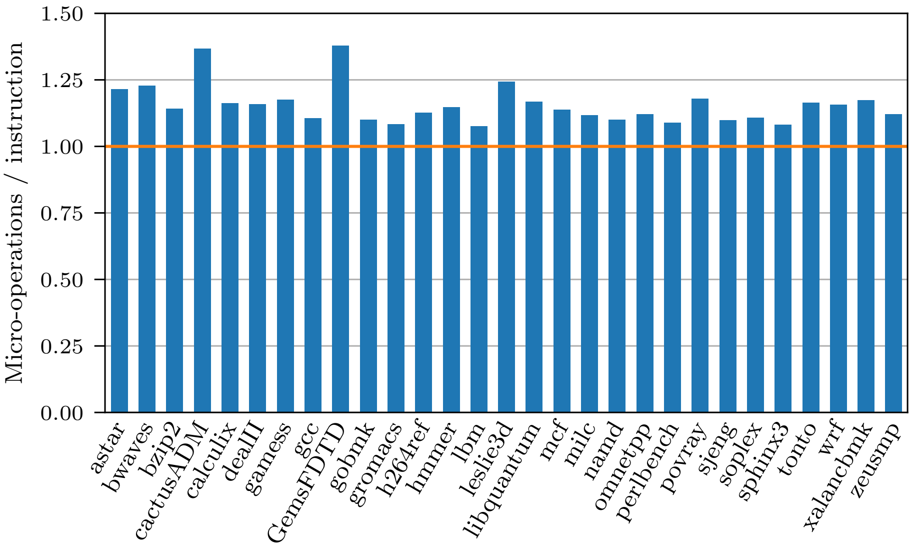


Figure 3.1: Number of micro-operations per instruction for all SPEC CPU 2006 benchmarks.

relatively small at 6.8%, but in the second case, the error would be as high as 38%.

3.3 Instruction Dependences

To execute an instruction on a functional unit, its operands need to be ready. When an instruction needs the result of a previous instruction as an operand, we say the current instruction depends on the previous. This type of dependence is very common in a program's instruction stream as illustrated by Example 3.1 which calculates the sum of a three-element vector. Figure 3.2 shows the corresponding data dependence graph. The numeric subscript indicates the number of times an instruction has been executed. If there were no dependences between the instructions, this code fragment, containing sixteen instructions, could be executed in four cycles on a four-wide superscalar processor. However, due to the dependences and assuming unit-latency instructions, it takes at least six cycles to execute. This illustrates the importance of taking into account instruction dependences when attempting to predict performance.

Instructions a through c initialize $R0$, $R1$ and $R2$ and are executed once. The purpose of these registers is to point at the memory location where the final result will be saved, to save the intermediary result of the sum and to point at address of the first element of the vector, respectively. Instruction d loads an element from the vector followed by instruction e which calculates the sum of that element and the previous value of $R1$. Instruction f increments the address in R_2 and serves as a loop counter. Instruction g checks whether we

```
        MOV    0xFC → R0          (a)
        MOV    0x00 → R1          (b)
        MOV    0xF0 → R2          (c)
  L1:   LD     [R2] → R3          (d)
        ADD    R1, R3 → R1        (e)
        ADD    R2, 0x04 → R2      (f)
        BNE    R2, 0xFC → L1      (g)
        ST     R1 → [R0]          (h)
```

Example 3.1: Code illustrating impact of instruction dependences.

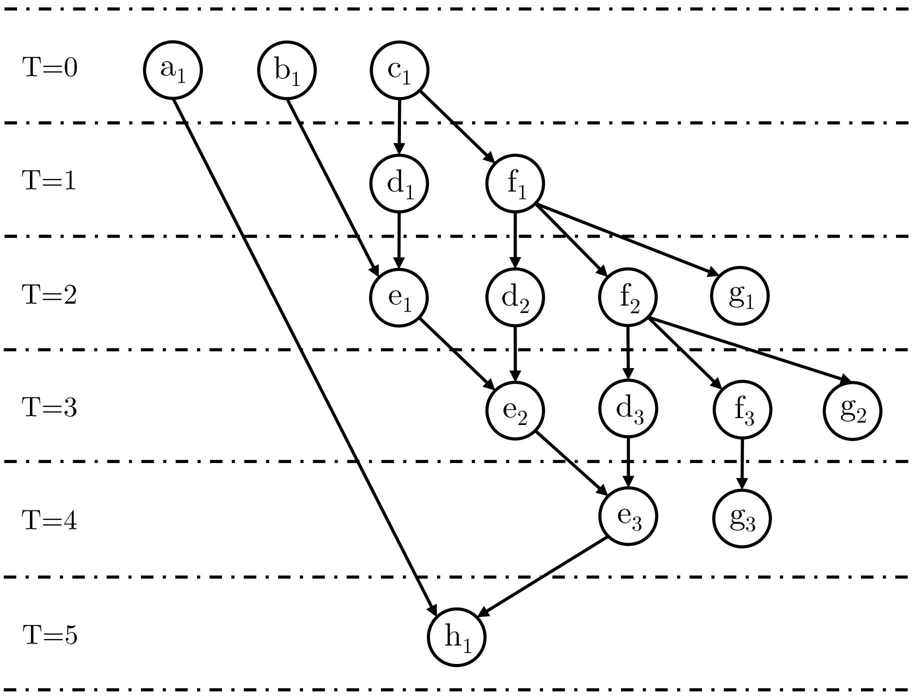


Figure 3.2: Corresponding data dependence graph for Example 3.1.

have executed the loop for all vector elements. The first two times we execute instruction g, $R2$ is not equal to $0xFC$ and thus we jump back to instruction d. After executing instructions d through g three times, we leave the loop and execute instruction h, which saves the final result at the address stored in $R0$.

To calculate performance within an interval we characterize a program's instruction dependences in different ways. We are interested in three different statistics called the *average path (AP)*, the *average branch path (ABP)* and the *critical path (CP)*. The average path measures the average number of producing instructions for all instructions, while the average branch path only takes producing instructions leading to a branch instruction into account. The critical path accounts for the longest dependence chain of producing instructions

```
1   buffers = 0, branch_buffers = 0
2   $AP_{sum} = 0$, $ABP_{sum} = 0$, $CP_{sum} = 0$
3   for instruction $i$ in instruction stream:
4       append instruction $i$ to buffer $B$
5       for all instructions in buffer $B$:
6           calculate # producing instructions
7       $AP_{sum} += \frac{\sum_{\forall i}(\#\, producing\ instructions)}{buffer\ size}$
8       if #branches > 0:
9           $ABP_{sum} += \frac{\sum_{\forall branch}(\#\, producing\ instructions)}{\#\, branches}$
10          increment $branch\_buffers$ counter
11      $CP_{sum} += \forall i\ max(\#\, producing\ instructions)$
12      remove first instruction from buffer B
13      increment $buffers$ counter
14  $AP = \frac{AP_{sum}}{buffers}$
15  $ABP = \frac{ABP_{sum}}{branch\_buffers}$
16  $CP = \frac{CP_{sum}}{buffers}$
```

Algorithm 3.1: Algorithm for calculating instruction dependence chains.

in an ROB only, rather than all chains. Producing instructions are defined as instructions that write to a register which is subsequently read by another instruction. For example, b_1 is a producing instruction for e_1 since it writes a value in R_1. e_1 is a producing instruction for e_2, also using register R_1, while b_1 and e_1 form a chain of producing instructions leading up to e_2.

AP, ABP and CP are used to calculate P_{hLLC}, c_{res} and D_{eff}, respectively, as we will explain later. All three of these statistics are related to the processor's ROB size. After all, instructions that are not part of the same ROB-size interval of instructions in the dynamic instruction stream are definitely not executing at the same time and will thus never have to wait on each other.

When an application is executing and the instructions flow through the ROB, the data dependence graph changes and so do the AP, ABP and CP metrics. Algorithm 3.1 shows how to calculate the dependence chains. While processing the instruction stream, a buffer B of instructions is maintained. For each instruction in that buffer we calculate how many producing instructions, i.e., instructions upon which it depends directly or indirectly, precede it. In line 7, we calculate AP by averaging the number of producing instructions for each instruction. Line 9 calculates ABP by averaging the producing instructions for branch instructions only, provided there are branch instructions in the buffer. For CP, calculated in line 11, we check which instruction has the highest number of producing instructions in its dependence chain. Subsequently, we remove the first instruction from the buffer, add a new one and recalculate the dependence statistics. In lines 14, 15 and 16, we average the calculated chains lengths across all buffers that are constructed from the instruction stream. For ABP we divide by the number of buffers containing at least one branch. Thus, the algorithm for calculating the dependence chains has a complexity of $O(N \cdot B)$, where N is the number of instructions and B the ROB size. Note that, to calculate the dependence chains for an ROB, B is preferably is at least as large as the ROB to prevent extrapolation.

Figure 3.3 provides a visualization of how this algorithm works for an ROB size of 8 instructions based on Example 3.1. The first row shows the instructions the same way they are displayed in the data dependence graph in Figure 3.2. The instructions are dispatched into the ROB from left to right. The second row shows the depth of the dependence chain. If the number below an instruction is 0, the instruction has executed and was committed. Note that, e.g., for the addition-instruction e_1, there are two different chains leading up to it, but we only keep track of the longest chain.

For the ROB containing the first eight instructions, the average path equals:

$$AP = \frac{1+1+1+2+3+2+3+3}{8} = \frac{16}{8} = 2 \tag{3.2}$$

There is only one branch in that ROB, g_1, so the average branch path equals 3. The longest instruction dependence chain, also called the critical chain, counts 3 instructions. Removing the first instruction from the ROB and inserting the second leads to an average path of $\frac{19}{8} = 2.375$. The average branch path does not change, but the critical path is now 4 because of instruction e_2 entering the ROB. After instructions a_1, b_1 and c_1 have been committed, which is the fourth row in Figure 3.3, d_1 resides at the head of the ROB. c_1 was a producing instruction for a whole sub-tree of instructions. Hence, the dependence counter for all of the depending instructions is decremented. The average path and critical path are again equal to 2 and 3, respectively. Furthermore, there are two branches in the ROB, g_1 and g_2, with a respective dependence chain length of 2 and 3, leading to an ABP of 2.5. Continuing these calculations for all shown ROBs and averaging them out gives us following results:

$$AP = \frac{2+2.375+2.625+2+2.125+2.5+1.75+2+2.25}{9} = 2.18 \tag{3.3}$$

$$ABP = \frac{3+3+3+2.5+2.5+2.5+1.5+2.5+2.5}{9} = 2.56 \tag{3.4}$$

$$CP = \frac{3+4+4+3+3+4+3+3+4}{9} = 3.44 \tag{3.5}$$

Figure 3.4 shows the measured AP, ABP and CP for one billion instructions of the SPEC CPU 2006 benchmarks with an ROB of size 128. All three metrics are significantly different in magnitude and vary across benchmarks, showing we cannot just replace one with another. For example, the average path is on average 2.9 times shorter than the critical path while the average branch path ranges from 50% shorter than the average path to 40% longer.

In the original interval model, the dispatch rate is set to the physical dispatch width D of the processor, assuming the reorder buffer (ROB) is large

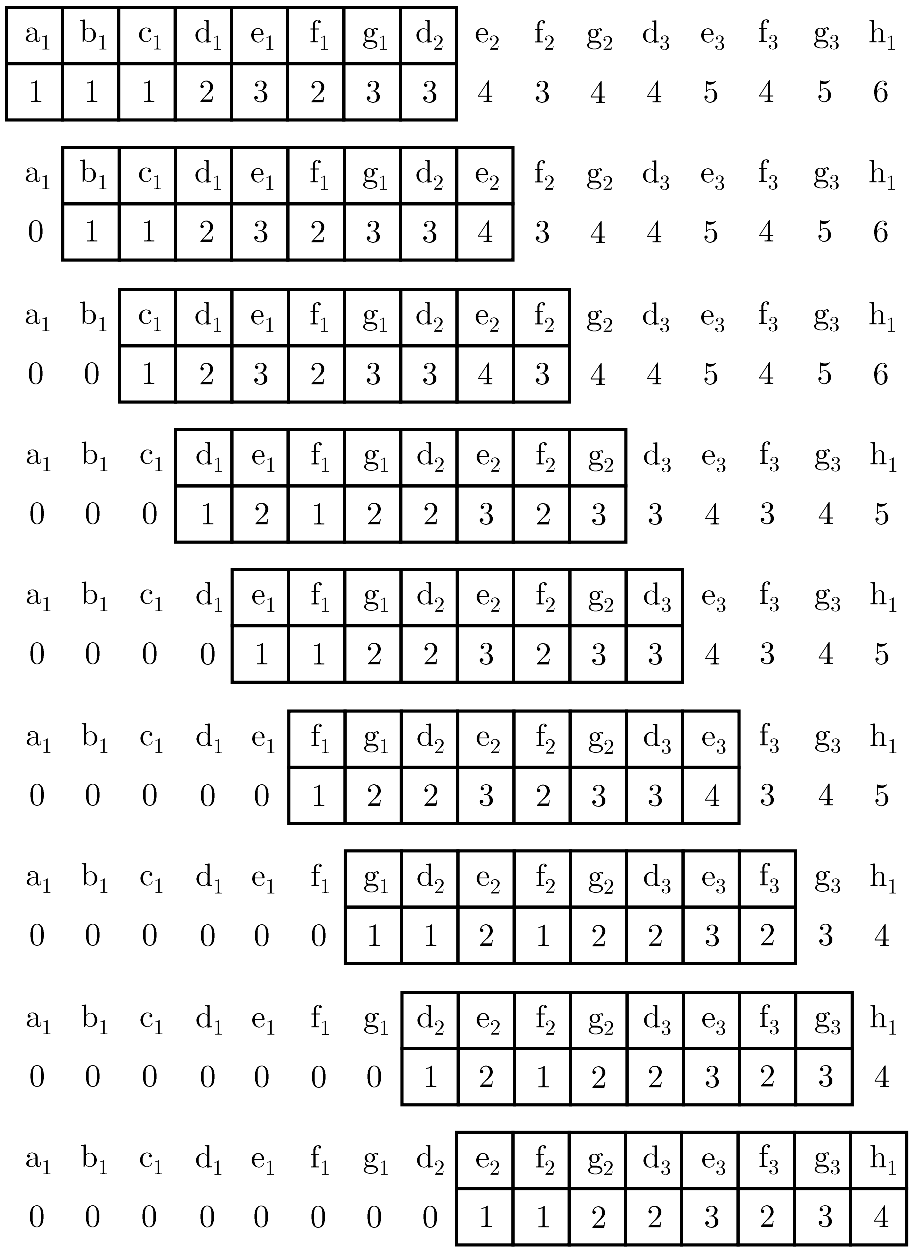

a1 b1 c1 d1 e1 f1 g1 d2 e2 f2 g2 d3 e3 f3 g3 h1
1 1 1 2 3 2 3 3 4 3 4 4 5 4 5 6
a1 b1 c1 d1 e1 f1 g1 d2 e2 f2 g2 d3 e3 f3 g3 h1
0 1 1 2 3 2 3 3 4 3 4 4 5 4 5 6
a1 b1 c1 d1 e1 f1 g1 d2 e2 f2 g2 d3 e3 f3 g3 h1
0 0 1 2 3 2 3 3 4 3 4 4 5 4 5 6
a1 b1 c1 d1 e1 f1 g1 d2 e2 f2 g2 d3 e3 f3 g3 h1
0 0 0 1 2 1 2 2 3 2 3 3 4 3 4 5
a1 b1 c1 d1 e1 f1 g1 d2 e2 f2 g2 d3 e3 f3 g3 h1
0 0 0 0 1 1 2 2 3 2 3 3 4 3 4 5
a1 b1 c1 d1 e1 f1 g1 d2 e2 f2 g2 d3 e3 f3 g3 h1
0 0 0 0 0 1 2 2 3 2 3 3 4 3 4 5
a1 b1 c1 d1 e1 f1 g1 d2 e2 f2 g2 d3 e3 f3 g3 h1
0 0 0 0 0 0 1 1 2 1 2 2 3 2 3 4
a1 b1 c1 d1 e1 f1 g1 d2 e2 f2 g2 d3 e3 f3 g3 h1
0 0 0 0 0 0 0 1 2 1 2 2 3 2 3 4
a1 b1 c1 d1 e1 f1 g1 d2 e2 f2 g2 d3 e3 f3 g3 h1
0 0 0 0 0 0 0 0 1 1 2 2 3 2 3 4

Figure 3.3: Number of producing instructions for Algorithm 3.1 for an ROB of 8 instructions.

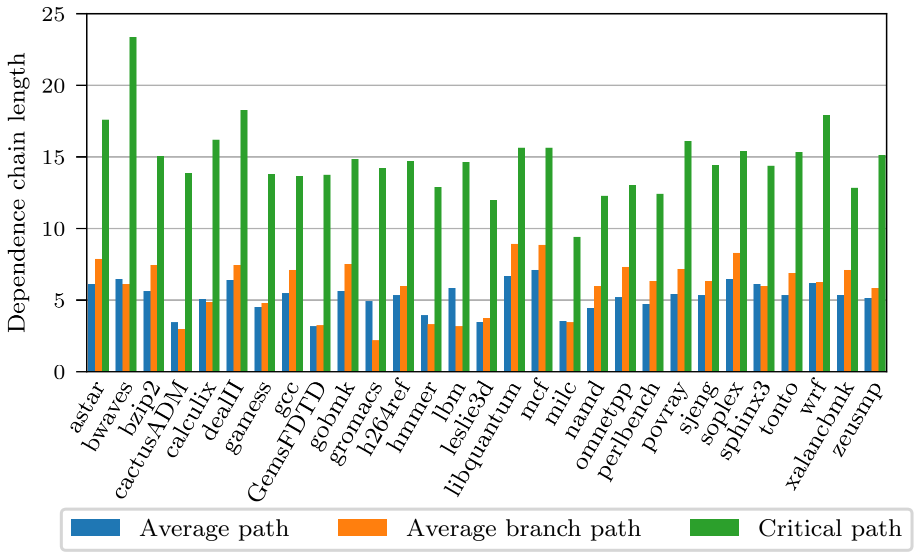


Figure 3.4: Comparison of the average path, average branch path and critical path for an ROB size of 128 for the SPEC CPU 2006 benchmarks.

enough to sustain an overall IPC of D (balanced design). However, we find that since the x86 architecture offers fewer architectural registers compared to the Alpha architecture and instruction latencies are higher, the dependence paths through register and memory dependences tend to take longer to resolve. This causes the achievable effective issue rate (and also commit rate) to be lower than the physical dispatch width. As the ROB and instruction queue fill up, the rate at which instructions can actually be dispatched decreases to what we call the effective dispatch rate.

The first step in modeling this effect is to calculate the average number of instructions that can be dispatched, issued and committed in a cycle. We call this the average number of independent instructions, $I(ROB)$[1]. We calculate $I(ROB)$ as shown in Equation 3.6:

$$I(ROB) = \frac{ROB}{lat \times CP(ROB)} \tag{3.6}$$

Here, $CP(ROB)$ is the critical path length for a fixed ROB size, while lat is the average instruction execution latency, including short (L1 and L2) load data cache misses. The reason for using the critical path rather than the average path to calculate $I(ROB)$ is that the ROB acts like a queue, meaning that the instructions leave the ROB in the order they entered it. Using the average path as the divider is correct to calculate the average number of independent instructions in the complete ROB but leads to an overestimation of the number of instructions that can actually leave the ROB every cycle.

[1]Independent instructions are instructions that do not have producing instructions in the current ROB.

After calculating $I(ROB)$, we replace the dispatch rate D in Equation 2.3 with the effective dispatch rate D_{eff}. This effective dispatch rate is calculated using Little's law, as follows:

$$D_{\mathit{eff}} = \min\left(D, \frac{ROB}{lat \cdot CP(ROB)}\right) \tag{3.7}$$

To further illustrate this intuitive equation, we apply it to the example from Example 3.1. Assuming a four-wide, superscalar processor with an ROB of 16 entries, the critical path is the path to instruction h_1, which is 6 instructions long. Assuming unit-latency instructions, the effective dispatch rate is 2.67 according to Equation 3.8.

$$D_{\mathit{eff}} = \min\left(4, \frac{16}{1 \cdot 6}\right) = 2.67 \tag{3.8}$$

Hence, executing those 16 instructions takes 6 cycles according to the interval model, which we already showed to be correct using the data dependence graph in Figure 3.2.

$$C = \frac{N}{D_{\mathit{eff}}} = \frac{16}{2.67} = 6 \tag{3.9}$$

3.4 Issue Stage Modeling

Current processors typically have multiple functional units for executing different instructions in parallel, of which several may be connected to a single port. For example, in the Intel Nehalem processor[2], there are only 6 ports serving 15 functional units as shown in Figure 3.5. If multiple instructions are to go to the same port in the same cycle, they need to be issued sequentially instead. Furthermore, if a non-pipelined functional unit is occupied, no new instructions of that type can be issued to that functional unit, even if the port is available. This has an important impact on performance, which we include in the model for improved accuracy.

We model the penalty introduced by a limited number of functional ports using a histogram of the instruction types in an application. We implement an algorithm that builds an issue schedule based on the frequencies of instructions in a histogram. We first check which instructions have to pass through one specific port. In the case of the Nehalem-processor displayed in Figure 3.5, this would be the load, store, divide, branch and some floating point instructions. The reasoning behind this is invariable of when they are issued, these instructions generate activity on that specific port. Afterwards, we loop over the other instruction categories that can be processed by multiple ports. For each of these categories, we take the already scheduled activity into account, and if it leads to better performance, split the execution of a certain instruction category over multiple functional ports. For example, an addition instruction is processed by the *Int ALU* units and could be scheduled on ports 0, 1 or 5.

[2]URL: `http://www.hardwaresecrets.com/inside-intel-nehalem-microarchitecture/4`

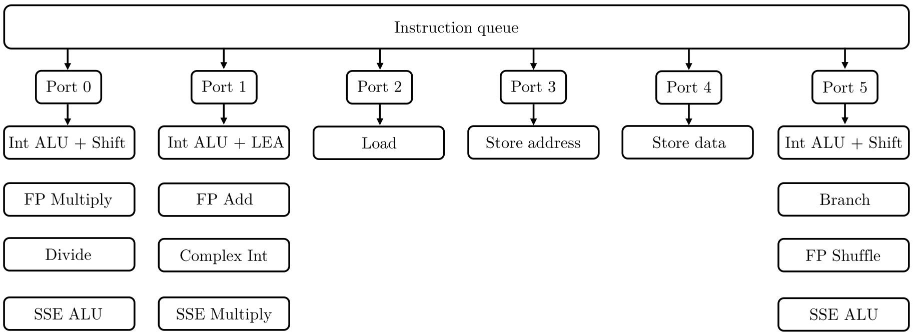


Figure 3.5: Nehalem issue stage with multiple functional units connected to one functional port.

Hence, we split its activity over these three ports as balanced as possible. Note that this algorithm uses a greedy approach and proposes an optimal issuing schedule, but does not necessarily build the schedule that a real processor will use. It does however provide a good approximation for the penalties introduced by waiting for an occupied port. Thus, assuming there is a mix of N instructions that have to be executed by the functional units connected to, e.g., port 0, it will take N cycles to forward them to their respective functional units, despite possibly having multiple available functional units. Hence, the effective dispatch rate for this case is $\frac{N}{N_p}$.

When modeling functional unit contention, we make a distinction between pipelined and non-pipelined functional units. For pipelined units, if there are N_i instructions of type i, and U_i functional units of that type, issuing them takes at least $\frac{N_i}{U_i}$ cycles. For non-pipelined units with latency lat_j, the minimal time to issue them equals $\frac{N_j \cdot lat_j}{U_j}$.

Thus, since both the number of functional units and the number of functional ports can further limit the effective dispatch rate, we rewrite Equation 3.7 as follows:

$$D_{eff} = \min\left(D, \frac{ROB}{lat \cdot CP(ROB)}, \frac{N}{N_p}, \frac{N \cdot U_i}{N_i}, \frac{N \cdot U_j}{N_j \cdot lat_j}\right) \tag{3.10}$$

in which N is the total number of micro-operations to execute, p ranges over all ports, i ranges over all types of pipelined functional units, and j over all non-pipelined functional units.

To explain the intuition behind this equation, we show two examples using the instruction mixes from Table 3.1 that can limit the effective dispatch rate. We assume a Nehalem-style processor similar to Figure 3.5. The ALU and branch instructions can be executed in 1 cycle, all loads and stores hit in the upper cache levels resulting in an average latency of only 2 cycles. The floating-point multiplications can be executed in 5 cycles. The functional unit that executes division instructions is the only non-pipelined unit and takes 5 cycles per division. This results in an average latency of 2 cycles for both instruction mixes. Furthermore, the processor has a physical dispatch width of 4, an ROB of 64 entries and the critical path is 8 instructions long.

Type	Amount	Latency
Load	40	2
Store	20	2
Int ALU	20	1
FP Multiply	10	5
Branch	10	1

Type	Amount	Latency
Load	40	2
Store	20	2
Int ALU	20	1
Divide	10	5
Branch	10	1

Table 3.1: Two examples of instruction mixes resulting in limitations on the effective dispatch rate due to a limited number of ports and functional units.

For the first instruction mix, the scheduling algorithm sends all load instructions to port 2 and all stores to ports 3 and 4 which results in port activity factors of 40 and two times 20, respectively. The branch and floating-point multiplication instructions are scheduled on ports 0 and 5. The ALU instructions are then divided over port 0 and 1. Hence, we arrive at the following vector for the scheduled activity on all ports: [15, 15, 40, 20, 20, 10]. Inserting this data in Equation 3.10 results in an effective dispatch rate of 2.5:

$$D_{\mathit{eff}} = \min\left(4, \frac{64}{2 \cdot 8}, \frac{100}{40}, \frac{100 \cdot 1}{40}\right) = \min(4, 4, 2.5, 2.5) = 2.5 \tag{3.11}$$

Note that only the maximum activity factor for the ports and functional units is included for simplicity as this will result in the minimum effective dispatch rate. We also omitted the last factor from Equation 3.10 since there are no instructions sent to non-pipelined functional units.

The second instruction mix results in the exact same vector for scheduled port activity. However, because the division function unit is not pipelined, the processor experiences extra contention resulting in a lower effective dispatch rate of 2:

$$D_{\mathit{eff}} = \min\left(4, \frac{64}{2 \cdot 8}, \frac{100}{40}, \frac{100 \cdot 1}{40}, \frac{100 \cdot 1}{10 \cdot 5}\right) = \min(4, 4, 2.5, 2.5, 2) = 2 \tag{3.12}$$

Thus, since executing N micro-operations takes $\frac{N}{D_{\mathit{eff}}}$ cycles, this equals 40 and 50 cycles for the first and second instruction mix, respectively. Intuitively, this makes sense. In the first instruction mix, the number of loads is the limiting factor, and with the load unit being pipelined, the port that provides the connection forms the limitation. In the second instruction mix, because the division unit is not pipelined, the number of divisions slows down the program even more than the load port does.

Figure 3.6 visualizes the factors that limit $\mathrm{D_{eff}}$ for an experiment containing one billion representative instructions from the SPEC CPU 2006 benchmarks. The most limiting factor for the base performance of each benchmark is represented by the lowest bar, following Equation 3.10. Here, *Dispatch* refers to the

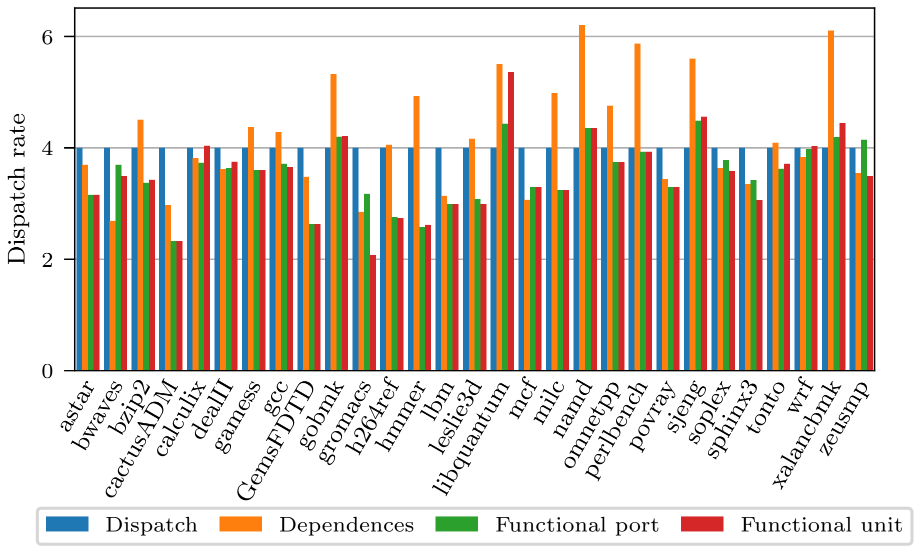


Figure 3.6: The four factors limiting the effective dispatch rate: the dispatch width, the critical path via inter-instruction dependences and the number of functional ports and functional units of a particular type.

physical dispatch width or first term in the equation while *Dependences* visualizes the critical path or second term. *Functional port* shows the third term and *Functional unit* combines the limitations of pipelined and non-pipelined functional units from the fourth and fifth term.

For most benchmarks, including *astar*, *cactusADM* and *gromacs*, the third or fourth bar are the lowest, which means that the effective dispatch rate is limited by either the functional ports or number of available functional units. Most often, we find this to be the result of a high fraction of loads or a significant number of divide instructions. For other benchmarks however, e.g., *bwaves* and *mcf*, the second bar is the lowest, which implies that inter-instruction dependences limit the effective dispatch rate. This typically occurs when the critical path is so long that it fills up the ROB. If neither of these factors limit the effective dispatch rate, the dispatch rate will be equal to the dispatch width, in which case the leftmost bar is the lowest. This is the case for *gobmk*, *namd*, *libquantum*, *sjeng* and *xalancbmk*. Note that the latter leads to the most optimal execution time.

To evaluate the accuracy of the proposed equation, we simulate a 'perfect' processor using Sniper [17]. This implies that no miss events occur in the processor. Hence, the branch predictor always predicts the direction of the branch correctly and all load and store instructions hit in the first level of the TLBs and cache hierarchy. The performance obtained from these simulations is the processor's maximum obtainable performance and should be comparable to the base component from our model.

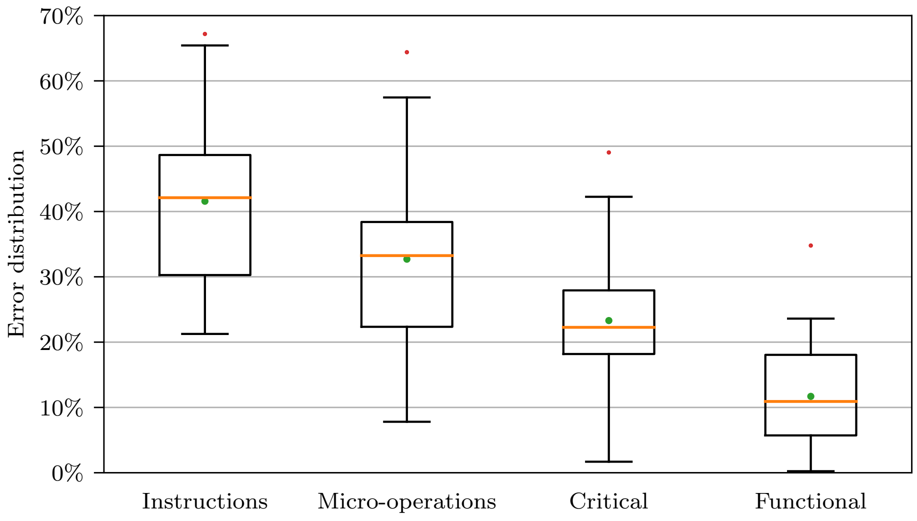


Figure 3.7: Decrease in prediction error due to the different components for the effective dispatch rate modeling when comparing against Sniper simulations without miss events for all SPEC CPU 2006 benchmarks.

Figure 3.7 shows the decrease in prediction error due to the addition of extra constraints to the effective dispatch rate as a box-and-whiskers plot. The box is the range between the first and third quartile, the horizontal line in the box is the mean, the dot is the median, the whiskers cover all points up to the 99th percentile, and the points represent the outliers. Each box-plot is built using the prediction errors for the performance of all SPEC CPU 2006 benchmarks.

The *Instructions* box-plot shows the error when calculating the base component by dividing the number of instructions with the physical dispatch width. The average prediction error for the performance is 41.6%. Splitting the instruction stream into micro-operations improves the average error by 8.9% to 32.7% as visualized by the *Micro-operations* box. Taking the dependence chains into account when calculating the effective dispatch rates again improves the error significantly. The average error, shown in the *Critical* box is now 23.3%. The last box, denoted by *Functional*, combines the performance constraints due to the functional units and issue ports. Taking this into account improves the average error by 11.6% to 11.7%. Note that, besides the average error reducing, the range of the prediction error also decreases due to modeling the extra constraints to the effective dispatch rate. This experiment clearly shows that modeling the base component as a division of the number of micro-operations by the effective dispatch rate, in which we take dependences and contention at the issue stage into account, is required to arrive at accurate performance predictions for the base CPI component.

```
1   input = $ABP(ROB)$, $I(ROB)$, $N_i$, $D$
2   while $N_i \geqslant D$:
3       if $ROB_i + D \leqslant ROB$:
4           $N_i = N_i - D$
5           $ROB_i = ROB_i + D$
6       else:
7           $N_i = N_i - (ROB - ROB_i)$
8           $ROB_i = ROB$
9
10      $leave = min(I(ROB_i), D)$
11      $ROB_i = ROB_i - leave$
12
13  $c_{res} = lat \cdot ABP(ROB_i)$
```

Algorithm 3.2: Algorithm for calculating the branch misprediction penalty.

3.5 Branch Predictor Modeling

The second term in the interval model equation calculates the penalty due to mispredicted branches. As described in Section 2.5.2, the penalty attributed to a branch misprediction can be split into two parts: the branch resolution time, c_{res}, and the front-end refill time, c_{fe}. The latter part of the penalty is a fixed penalty and depends solely on the number of stages in the processor's front-end. The former is dependent on the instruction dependences in the program.

In the original interval model, experiments pointed out that the branch instruction was more often than not the last instruction to execute, indicating it is on the critical path [30]. However, this assumption does not hold for x86-programs, likely due to the longer dependence chains as described in Section 3.3. The average branch path, as Figure 3.4 showed, is equal to the number of producing instructions leading to a branch instruction and is consistently shorter than the critical path. On average, across all SPEC CPU 2006 benchmarks, the average branch path is 2.8 times shorter. While it is possible that the average branch path is actually a subset in the critical path, attributing the full length of the critical path to the branch misprediction penalty does not lead to an accurate prediction.

Algorithm 3.2, called the 'leaky-bucket' algorithm devised by Michaud et al. [49], is used to calculate the branch misprediction penalty. The inputs are the average branch path, $ABP(ROB)$, and independent instructions, $I(ROB)$, which are functions of the ROB size (see Section 3.3), the number of instructions between two branch mispredictions, N_i, and the physical dispatch width, D. ROB_i is the current number of instructions in the ROB while ROB denotes its physical size. In each iteration, we calculate how many instructions can enter the ROB, which is either equal to the dispatch width (lines 4 and 5) or the number of free slots in the ROB (lines 7 and 8). We then calculate the number of instructions that can finish, based on the average number of independent instructions. When the iteration is finished, this is, when all useful instructions in an interval have been dispatched, we calculate the branch resolution time. This penalty is equal to the average instruction latency, *lat*, times the average branch path length for the number of instructions that reside in the ROB.

The most significant modification to this penalty term however is the way we obtain the branch misprediction rates. As stated in the previous chapter, the original interval model uses, among others, a branch predictor simulator to obtain branch misprediction rates. Because the goal of this work is to get faster and more accurate processor performance predictions, we need to obtain branch misprediction rates without simulation.

To achieve this, we rely on a metric called linear branch entropy [22][3]. The reason why branches can be predicted is that, in general, they are executed many times and branch outcomes are often correlated. Depending on the previous outcome of a branch, it can have a higher probability to be taken or not. Algorithm 3.3 shows an example of a predictable and unpredictable branch.

```
1  for i: 0 → 100
2      if i mod 2 = 0: // branch 1
3          if random < 0.5: // branch 2
4              ... perform if calculation ...
5          else:
6              ... perform else calculation ...
7      else:
8          ... perform else calculation ...
```

Algorithm 3.3: Code with a predictable and unpredictable branch.

In the case of branch 1 on line 2, we will execute the code inside the 'if' statement when variable i modulo two equals 0, this is when i is divisible by two. Hence, in that case, the branch is taken, otherwise it is not. This branch is perfectly predictable by a branch predictor with one bit of local branch history. After all, if the branch predictor knows that the previous branch was taken, it will know that the current one should not be taken. Branch 2 on line 3, however, is completely unpredictable because it relies on a random number between 0 and 1. The current random number is in no way correlated to the previous one, hence no matter how large the branch history is, the branch predictor will never be able to recognize a pattern and can thus never predict the branch direction accurately.

This predictable or non-predictable nature of branches is related to a physical metric called linear branch entropy. Entropy captures the disorder in a system and can be used to summarize whether there is a pattern in the measured data. Since the way a branch predictor works is by acting on taken/not-taken patterns from previous branches, using entropy to describe such a pattern is a good fit.

[3]This work was performed by a fellow PhD-student with whom I collaborated.

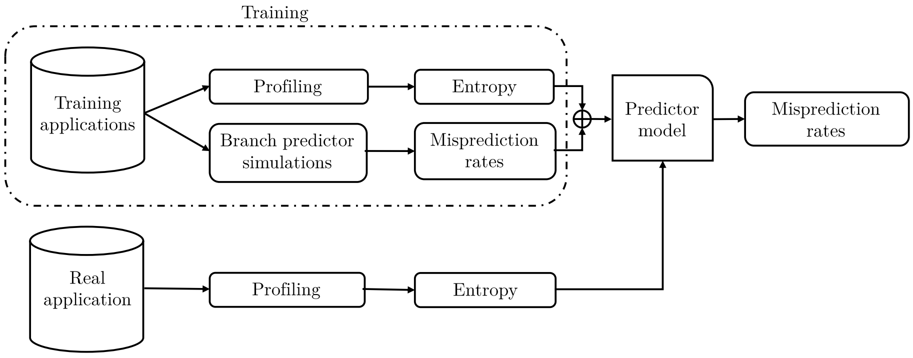


Figure 3.8: Visualization of the training step and application profiling to obtain branch misprediction rates.

For each static branch b and history pattern H, a record is kept for the number of taken, $T(b,H)$, and not-taken branch outcomes, $NT(b,H)$. The probability for a branch to be taken, given a specific history pattern, is shown in Equation 3.13. This probability is used to define the linear branch entropy in Equation 3.14.

$$p(b,H) = \frac{T(b,H)}{T(b,H)+NT(b,H)} \tag{3.13}$$

$$E(p) = 2 \cdot \min(p, 1-p) \tag{3.14}$$

Equations 3.13 and 3.14 calculate the entropy for one specific branch and one history pattern. However, for calculating performance in the interval model we are interested in the overall number of branch misses. Therefore, we average all the different branch entropy numbers across all branches and all history patterns in Equation 3.15 for a fixed history length. Here, $n(b,H)$ is the number of times a branch b is executed with history pattern H and N_b is the total number of dynamically executed branches.

$$E = \frac{1}{N_b} \sum_b \sum_H n(b,H) \cdot E(p(b,H)) \tag{3.15}$$

The average branch entropy needs to be transformed into a number of branch mispredictions. In order to achieve this, De Pestel et al. [22] propose a framework to build a model that connects branch entropy to branch misprediction rates for specific branch predictors. This framework is schematically shown in *Figure 3.8.* From a set of training applications, entropy numbers and branch misprediction rates are gathered using profiling runs and simulations, respectively. These are used to build a linear model for a specific branch predictor.

An example of such a linear model can be found in Figure 3.9, where a linear fit for branch entropy and branch predictor miss rates for a 4KB GAg branch predictor is shown. The branch entropy of an application of interest is then used as input to this model to predict the number of branch mispredictions

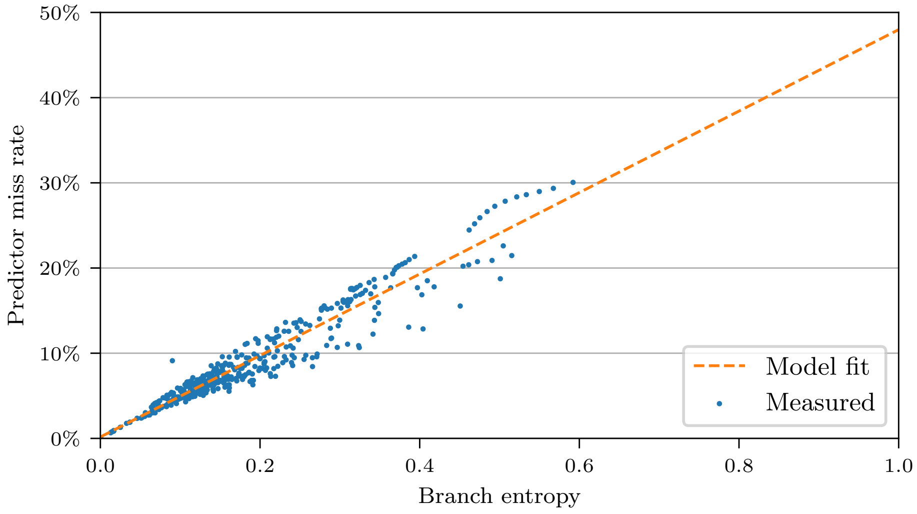


Figure 3.9: Linear fit for branch entropy and missprediction rates for more than 400 experiments.

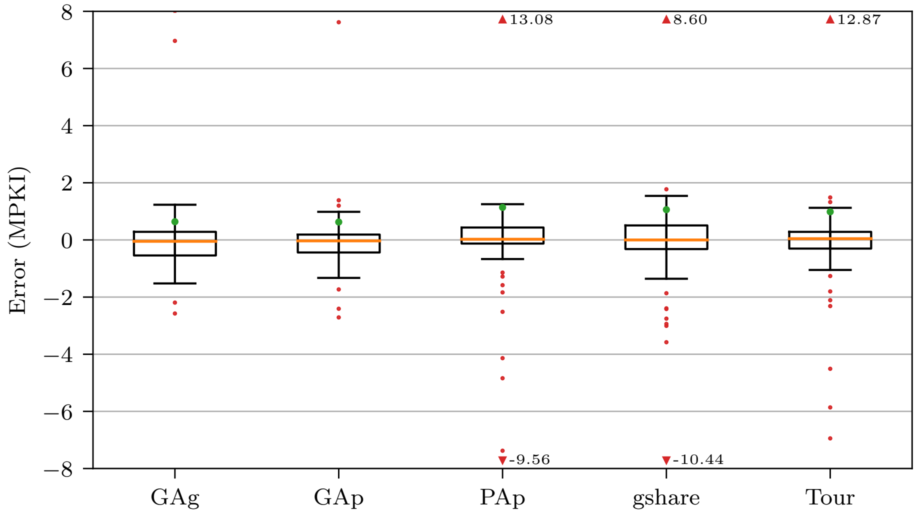


Figure 3.10: Box-and-whiskers plot for five different branch predictors showing the prediction accuracy.

for a specific branch predictor. Compared to the previous interval model, the advantage is that the initial simulations are a one-time cost that never need to be repeated because the branch predictor model is suitable across different applications.

In Figure 3.10, based on the work of De Pestel et al. [22], we show the prediction accuracy for this approach for five different branch predictors: a GAg predictor, a GAp predictor, a PAp predictor, a gshare predictor, and a tournament predictor consisting of a GAp and PAp predictor. All predictors have a size of approximately 4KB. The average MPKI for all predictors is 9.3, 8.5, 7.6, 6.9, and 7.1, respectively. The absolute errors for the MPKI are 0.64, 0.63, 1.14, 1.06, and 0.99, respectively. The accuracy is shown as a box-and-whiskers plot where we plot the difference in MPKI (Misses Per Kilo Instructions) between simulation and the linear branch entropy model. Just like the previous box-plot, the box is formed by the first and third quartile, but the whiskers are now set at 1.5× the interquartile distance. All other points are outliers. Note that we show some outliers as a triangle point up or down with an annotated error indicated that they are outside of the y-axis range. Both for PAp and gshare there are a number of bigger outliers, which can be attributed to the execution of *gcc* using different input sets. The reason for the erratic behavior is that the benchmark executes many unique branches causing aliasing making it hard to predict. Part of the tournament predictor is a PAp, also leading to some outliers.

3.6 Core Power Modeling

Designing application-specific processors to improve energy efficiency requires a power model. To estimate power, we use the McPAT tool [46], which provides an XML-interface to supply all necessary inputs. To model the core power, it requires the configuration of the processor and the activity factors (i.e., the number of accesses) for each component[4].

The required parameters to describe the processor architecture include the physical dispatch, issue and commit width, the number of ALUs and the number of entries in the instruction queue and ROB. Furthermore, we supply McPAT with a description of all table sizes in the branch predictor. McPAT then combines this with the supplied processor frequency and supply voltage or uses an ITRS [6] default.

For our model, we deduce the activity factors from the analytical performance model, instead of measuring them in simulation. Many of the inputs are directly measured by the profiling tool for the performance model. The number of micro-instructions is used to estimate the amount of reads from and writes to both the physical register file, ROB structure, instruction queue and load-store queues. The instruction mix can also be combined with the predicted number of cycles to execute an application to deduce the activity factors for the functional units:

$$Activity\ Factor_i = \frac{\#Instruction_i}{Application\ Cycle\ Count} \tag{3.16}$$

where i stands for the pair of one specific functional unit and its associated instruction type.

[4]In Chapter 4, we will discuss the power modeling related to the memory requests performed by the processor.

The result of combining the architecture parameters with the activity factors is an estimation of the power consumption. Afterwards, energy consumption can be predicted by multiplying the predicted power consumption with the predicted execution time.

Chapter 4

Modeling the Memory Subsystem

4.1 Cache Hierarchy and the Interval Model

The cache hierarchy in a processor is used to store the most frequently used data close to the computation logic. This speeds up the processor's execution significantly. However, because caches have limited size, requests cannot always be satisfied since the data may have been removed from the cache in the meanwhile. Requests that do not find their data in the cache are called 'cache misses'. We distinguish three types of cache misses: cold, conflict and capacity misses.

Cold misses occur because it is the first time that a data block is requested by the processor. Cold misses are exclusively application-dependent and thus independent of the micro-architecture. Conflict misses ensue when two or more addresses are mapped to the same set in a cache. This results in one request removing the data from a previous one. Capacity misses occur because the physical size of the cache is not big enough to fit the application's data set. The latter two categories of misses are both dependent on the application and the processor's micro-architecture.

In Equation 3.1, terms three and four, quantify the impact of instruction and data cache misses. Similar to the interval model [32], the penalty for instruction cache misses is calculated as the number of misses at each level i multiplied by the access time to the next cache level $i+1$. The penalty for long-latency load misses (i.e., LLC load misses) equals the number of load misses in the LLC times the memory access time, $c_{mem} + c_{bus}$, divided by the amount of Memory-Level Parallelism (MLP). c_{bus} is the number of cycles spent on the memory bus, including waiting time for the memory bus if it is occupied (see Section 4.7). MLP equals the average number of overlapping misses if at least one is outstanding (see Section 4.3).

4.2 Cache Miss Rate Modeling

To estimate cache miss rates using a micro-architecture independent profile, we use the StatStack statistical cache model [28], which provides the miss ratio for fully-associative Least Recently Used (LRU) caches of arbitrary sizes[1].

Other cache replacement strategies could be modeled using ranking functions as described by Beckmann et al. [12]. Note however that their approach likely only works well for last-level caches as it requires the memory access pattern to be random. This assumption does not hold for the cache levels closest to the core as there is both temporal and spatial locality in a program's memory access pattern. Hence, to model different cache replacement policies for the complete cache hierarchy, it might be required to employ a hybrid strategy consisting of both stack distances and ranking functions for different cache levels.

StatStack uses the concept of reuse distances to estimate cache behavior. Reuse distances count the total number of memory accesses to other cache lines between two accesses to the same cache line. The reuse distances for an application are used to build a histogram of an application's reuse behavior, which is then transformed into a stack distance distribution. The stack distance distribution describes the number of unique cache lines accessed between two accesses to the same cache line. Hence, this distribution can be used to estimate the miss ratio for a fully-associative LRU cache by counting the number of accesses exhibiting a stack distance greater than the cache size. Because reuse distances only require keeping a counter, they are far cheaper to collect than stack distances, for which a complete stack is needed.

Figure 4.1 shows an example of a stream of unique memory addresses A through C. For each unique address, the reuses are connected. Below the address stream, we indicate what the reuse distance (RD) and stack distance (SD) is for each memory address. For example, the reuse distance between the first and second use of A is four, but the stack distance only amounts to 2. On the other hand, between the second and third use of A, the only intervening access is one to C, resulting in both the reuse and stack distance being equal to 1.

Profiling an application's reuse distance distribution can be done independently of the cache configuration. However, measuring reuse distances for all memory operations would introduce high overhead and thus low profiling speed. Therefore, StatStack reduces profiling overhead by collecting only a sample of the reuse distances in an application. Berg et al. [13] show that it is possible to profile an application to get its reuse distance distribution with very low overhead using hardware performance counters. This approach has been further optimized by Sembrant et al. [56].

Transforming reuse distances into stack distances follows a three-step algorithm. First, reuse distances are binned together into a reuse distance his-

[1]We collaborated closely with Moncef Mechri from Uppsala University to modify StatStack to our needs.

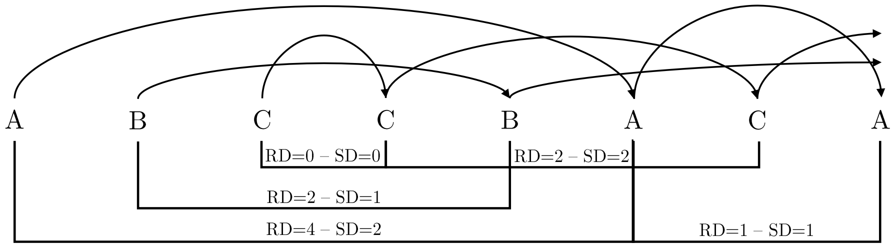


Figure 4.1: Memory address stream with indicated reuses and respective stack distances.

togram. Next for each bin in the histogram, the fraction of reuse distances greater than the current reuse distance is calculated. Based on these fractions, for each distinct reuse distance, the expected stack distance can be calculated as a running sum of reuse distances smaller than the current reuse distance. Graphically, in Figure 4.1, this means that for each arrow connecting a reuse, the number of intersecting arrows are counted as this approximates the stack distance. For example, for the first reuse of A, there are two intersecting arrows. Thus, the expected stack distance is 2.

For the interval model, StatStack was extended to differentiate between load versus store misses by building stack distance histograms for each memory access type separately. Note that the reuse distance histograms have to be built for both memory access types combined. The interval model does not model the performance impact store misses may have, i.e., it is assumed that the processor does not often stall on a store miss. However, store misses contribute to memory bandwidth contention and power consumption of the cache and core, which we do account for in the model.

Originally, StatStack modeled cache miss rates exclusively for the last-level cache. However, predicting processor performance requires the miss rates across the entire cache hierarchy. To achieve this, we estimate the miss ratios for each level in the cache hierarchy independently as if it was the only cache level. Essentially, we count the number of accesses which are bigger than the respective cache sizes in the stack distance distribution for each cache level separately. Note that this implicitly assumes that the smaller cache levels contain a subset of the data from the larger cache levels, and thus limits us to modeling inclusive cache hierarchies.

Figure 4.2 shows the predicted and simulated MPKI (Misses Per Kilo Instructions) for a three-level set-associative cache hierarchy where the cache level sizes are 32, 256 and 8192 KB, respectively[2]. Every odd bar shows the predicted MPKI, indicated with the StatStack label, and every even bar shows the simulated MPKI using the Sniper simulator. Most benchmarks have a negligible amount of cache misses. Naturally, the interval model will not suffer from large prediction errors if the number of misses is low, but estimated inac-

[2]Unless mentioned otherwise, all figures and numbers in the following sections are generated using these sizes for the cache hierarchy.

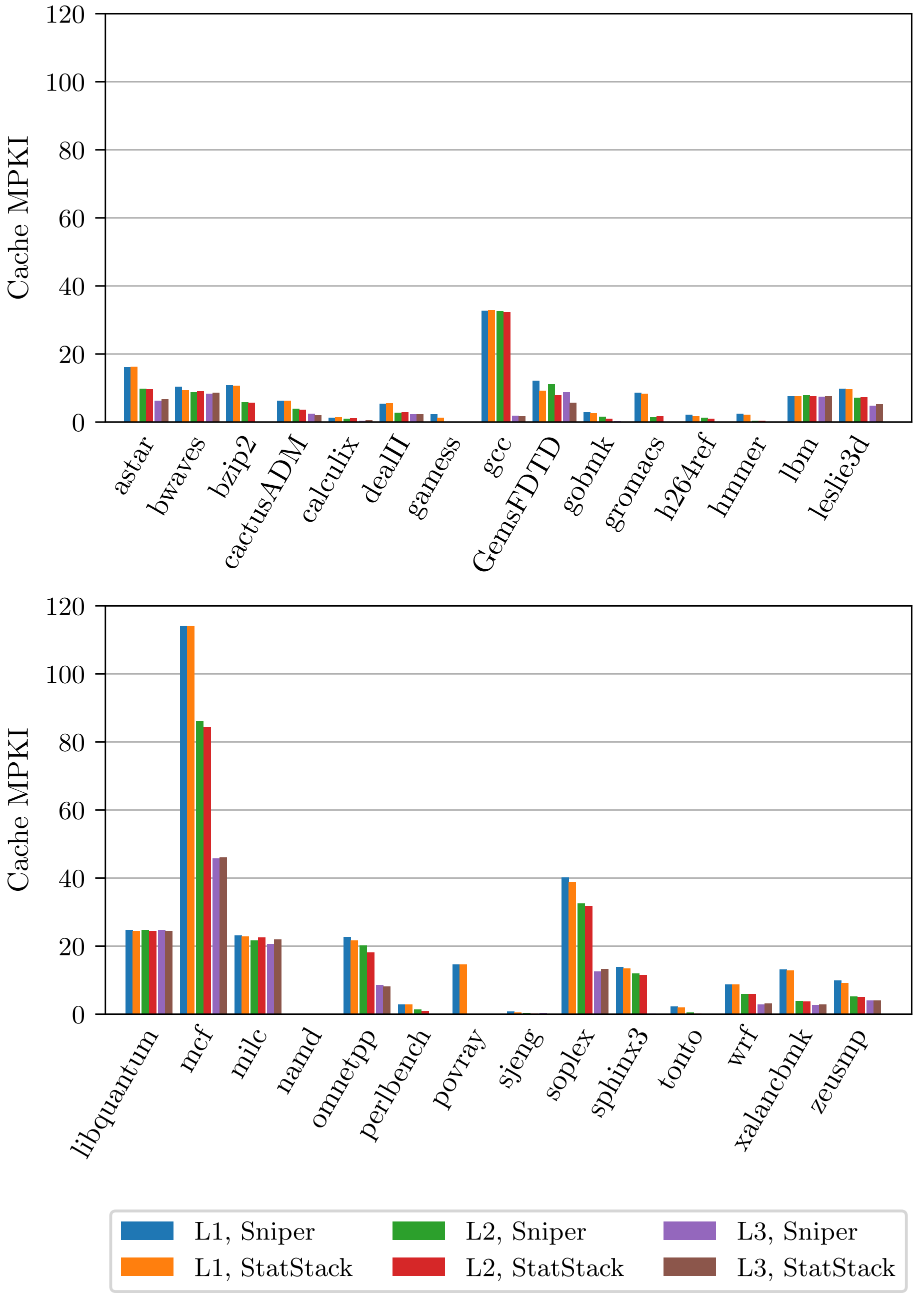


Figure 4.2: Estimated cache misses for a three-level cache hierarchy with sizes 32, 256 and 8192 KB.

curately. For the benchmarks with a significant number of misses (e.g., above 10 MPKI), the StatStack model performs well and has an average prediction error of 4.1%, 6.7% and 3.5%, respectively, for the three cache levels. Note that *GemsFDTD* is the one exception having a prediction error close to 30% while still having a non-negligible MPKI of 12. This experiment shows that modeling the cache levels separately provides good accuracy across a standard three-level cache hierarchy. Furthermore, it also shows that because the goal of a good hashing function is to spread the cache accesses uniformly over the sets, the approximation of assuming a fully-associative cache to model a set-associative cache is sensible.

The instruction cache behavior is modeled similarly as the behavior for the data caches. The reuse distance distribution is computed over the instruction address stream and afterwards transformed to stack distances.

4.3 Memory-Level Parallelism

Memory-level parallelism (MLP) is defined as the average number of main memory accesses (LLC misses) that can be processed in parallel, if at least one is outstanding [20]. This assumes that the processor cache hierarchy is non-blocking. Accesses to main memory can be sent over multiple DRAM-channels. Main memory typically consists of multiple DRAM banks, that can each process one access at a time. The interval model assumes that the penalty of multiple parallel accesses equals the penalty of a single access, explaining the division of the last term in Equation 3.1 by the MLP.

MLP has a non-negligible impact on performance, as illustrated in Figure 4.3. The leftmost bar shows normalized CPI stacks for detailed simulation using Sniper, with two components: execution time due to DRAM accesses (i.e., the memory component), and all other components, aggregated in 'CPI other'. The rightmost bar is normalized to Sniper's simulated execution time and represents the absence of MLP ($MLP = 1$), i.e., all memory accesses are serialized. The takeaway is that MLP has a significant impact on overall performance, hence modeling its impact is important. Not modeling MLP (i.e., assuming there is no MLP) leads to an average error of 24.6%, with a 96% maximum error.

MLP tends to exhibit a bursty behavior, which makes it difficult to predict. In this work, we discuss two different techniques to model MLP that leverage different insights with respect to memory behavior. Both of the techniques have their merits and drawbacks with respect to speed, accuracy and applicability.

To speed up simulation of an application, processor architects often opt to analyze a smaller representative part of an application, as described in Section 2.2. However, this introduces a difficult to solve problem, namely the presence of cold misses which would not exist if the application was simulated in its entirety. A lot of work focuses on eliminating these cold misses through prepending a warm up phase to the sample [26, 36] or leveraging reuse distances

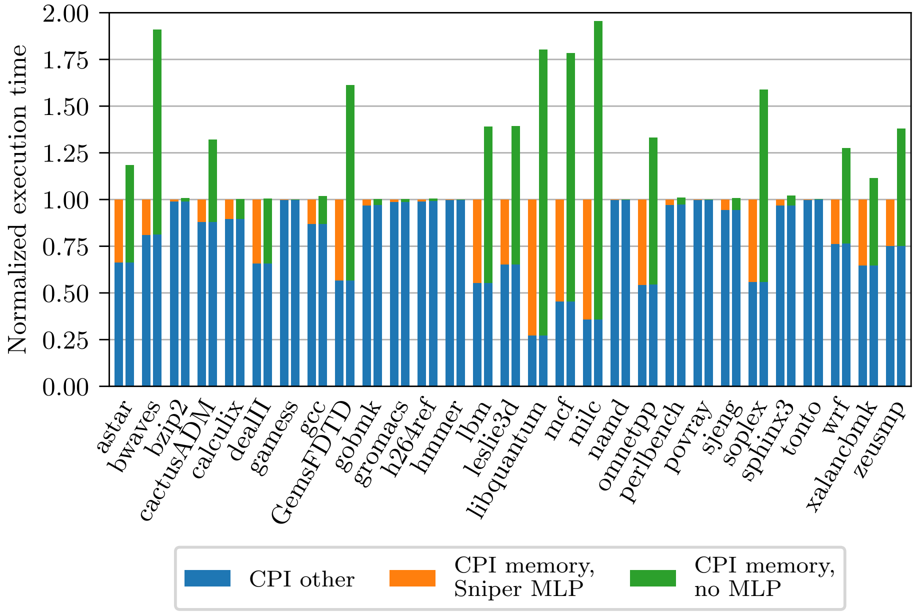


Figure 4.3: Normalized execution time with breakdown in memory cycles and other cycles for Sniper simulations (left bar) and when no MLP is modeled (right bar).

to estimate a cache state [52]. Note that we cannot use checkpointing, because checkpoints are inherently dependent on the processor's micro-architecture and our focus is on developing micro-architectural independent models.

Elimination of these cold misses can require a long and slow warm-up phase. Figure 4.4 shows the relative number of cold misses versus capacity misses for both loads and stores in two experiments. The left bar shows the breakdown of cold and capacity load and store misses for a cache simulation of a trace of 1 billion instructions. The right bar shows the same breakdown of misses but we prepend a cache warm-up of 1 billion instructions (for which we do not count the number of cache misses). We normalize the total number of misses to the number of misses in the first experiment. This shows that the total number of misses shrinks for about one third of the benchmarks, but stays similar for the others. The ratio of cold versus capacity misses does diminish for almost all benchmarks. For example, in the case of *bwaves*, almost all cold misses are eliminated. However, the warm-up does not eliminate the number of cold misses consistently for all benchmarks. Comparing the green and blue bar for, e.g., *cactusADM*, *mcf* and *milc*, shows us that a significant part of the load misses are still cold load misses. This leaves two options to model the memory hierarchy and more importantly, the MLP, accurately: either cold load misses are included as part of the modeling, or a larger warm-up phase is employed.

The first technique to model MLP leverages the presence of cold misses to model the bursty nature of the MLP. Since cold misses are included in the MLP

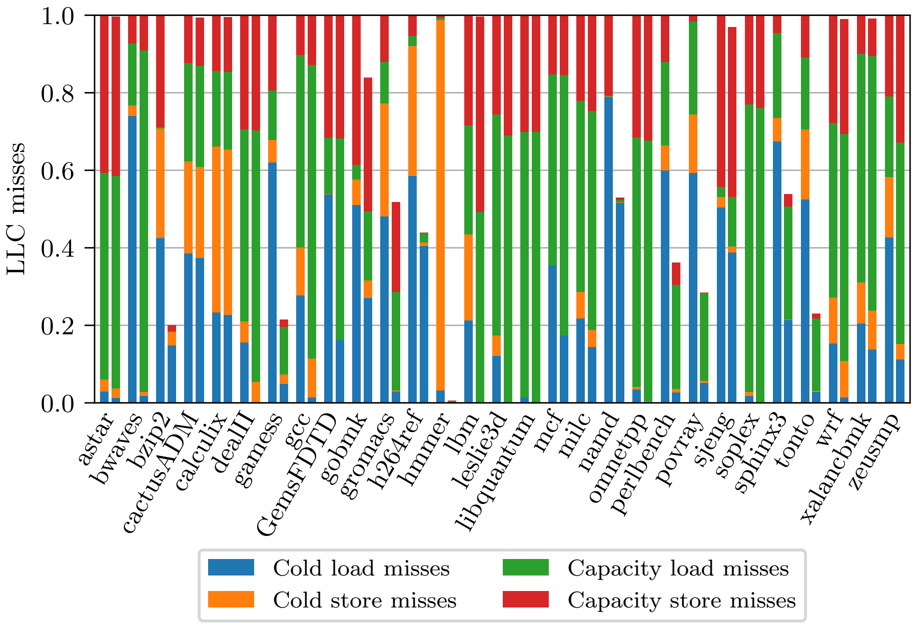


Figure 4.4: Breakdown of cold and capacity load and store misses for a trace of 1 billion instructions (left bar) and a trace of 2 billion instructions using 1 billion instructions as warm-up (right bar).

model, it is possible to employ this technique in combination with application traces without the need for a long cache warm-up phase. Because of its strong reliance on cold misses, we call this technique the *cold-miss MLP model*. Note that, in the absence of cold misses, this technique will not necessarily predict the MLP accurately as it cannot capture the burstiness of the MLP properly.

The second MLP prediction technique is a generalized technique that can be used without relying on the presence of cold misses. It leverages the fact that memory accesses often follow a strided access pattern. A strided access pattern is a pattern where subsequent accesses request data from memory locations that have a constant offset with respect to each other. These offsets can be profiled and summarized in distributions to figure out which accesses will miss possibly leading to multiple, simultaneous memory accesses. Because these distributions can be profiled using sampling, the profiling is 40% faster than profiling all memory accesses to collect cold miss distributions. We call this technique the *stride MLP model*.

It is important to realize that, if a significant portion of the load misses are cold load misses, the stride MLP model will not necessarily predict the MLP accurately. The reason for this is that StatStack samples memory accesses, making it impossible to confirm whether a specific access has been executed before. Hence, it cannot pinpoint the exact location of those cold misses. Since MLP is defined as the number of main memory accesses occurring in parallel, the locations of the accesses in the dynamic instruction stream are key. After all, if the locations of the accesses are unknown, it is impossible to gauge if they occur in parallel.

4.4 Cold-Miss MLP Model

Following interval analysis, the number of parallel memory accesses equals the number of independent LLC misses that occur within the ROB. The amount of MLP is thus dependent on micro-architectural features (the size of the ROB, size of the caches, number of MSHRs), as well as application characteristics (which instructions cause misses and how they depend on each other). However, in our micro-architecture independent profile, we only have limited information about these characteristics: we have the LLC miss rate from StatStack, but not the 'location' of the individual misses in the instruction stream, making it hard to estimate MLP; moreover, although we profile dependences between instructions, we do not know the dependences between LLC misses. Modeling memory behavior and MLP accurately is not straightforward and turns out to be one of the largest contributors to the total error of the model, see Section 6.2.

LLC misses frequently occur in bursts: when a load misses in the LLC, there is a large probability that loads nearby in the instruction stream will also miss in the LLC. As a result, assuming that LLC misses are uniformly distributed across the application leads to inaccurate MLP estimates. We find that, in the absence of a long cache warm-up phase and relatively short instruction samples of 1 billion instructions, LLC miss bursts are largely caused by cold misses, i.e., the first time a cache block is accessed. Capacity and conflict misses, i.e., the cache block was in the cache but has been evicted, are more uniformly distributed. The intuition is that throughout its execution, an application will load new data structures on which it will compute. This typically leads to bursts of cold misses. Conflict misses on the other hand, are caused by too many unique accesses to the same set in the cache. This occurs more spread across the application's execution, so there is less burstiness due to conflict misses.

Cold misses can be located using a micro-architecture independent profile by keeping track of the first access to a certain address. Because we have to check for every address if it has been accessed before, keeping track of all addresses leads to a large structure and high lookup times. To reduce this overhead, we assume a limited set of allowed cache block sizes (e.g., 32, 64 and 128 bytes), and we record only cold misses for these cache block sizes. The final profile consists of the distribution of the number of cold misses in an ROB, for different ROB and cache line sizes.

We leverage the following assumptions to estimate MLP:

- m_{LLC}^{cf}, m_{LLC}^{cold} and $m_{LLC}^{cold}(ROB)$ represent the number of capacity/conflict misses, the number of cold misses, and the average number of cold misses per ROB containing at least one cold miss, respectively.

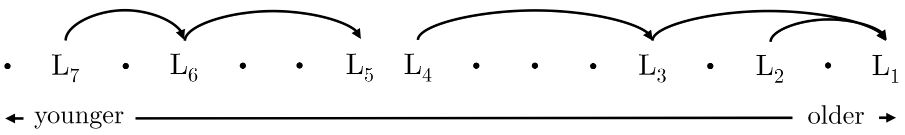


Figure 4.5: Example of a load dependence distribution.

- The load distribution $f(\ell)$ characterizes the dependences between loads. In this distribution, ℓ is the number of loads on the dependence path leading to a load in the ROB including that last load ($\ell = 1$ means that the load is independent of other loads), and $f(\ell)$ is the frequency of loads with ℓ loads on their dependence path. Figure 4.5 shows an example for a 16-entry ROB. The oldest instruction is located on the right and the arrows indicate dependences between loads. The ROB contains 7 loads; two of those loads appear at the head of a load dependence chain (L_1 and L_5 have $\ell = 1$); there are three loads that appear as the second load on a load dependence chain (L_2, L_3 and L_6 have $\ell = 2$); and there are two loads that appear as the third load on a load dependence chain (L_4 and L_7 have $\ell = 3$). Hence, the corresponding load distribution $f(\ell)$ equals $[\frac{2}{7}; \frac{3}{7}; \frac{2}{7}]$.

- M_{LLC} and M_{LLC}^{cf}, which denote the overall LLC miss rate and the capacity/conflict LLC miss rate, respectively. In the model, we use the miss rate as an approximation for the probability for a load to cause a cache miss.

- $\bar{L}(ROB)$ is the average number of loads per ROB, i.e., the fraction of loads in the instruction mix times the ROB size.

Our MLP model is split up into two parts: MLP due to cold misses and MLP due to capacity/conflict misses. The cold-miss MLP is the average number of independent cold misses in the ROB. A load miss that is the ℓ-th load in a dependence path will be an independent miss if all $\ell - 1$ previous loads on its path are not misses, which has a probability of $(1 - M_{LLC})^{\ell-1}$. From the $m_{LLC}^{cold}(ROB)$ cold misses in the ROB, $m_{LLC}^{cold}(ROB) \cdot f(\ell)$ are the ℓ-th load on a dependence path, so the number of independent cold misses in the ROB, i.e., the cold-miss MLP, can be estimated as:

$$MLP^{cold} = \sum_{\forall \ell} (1 - M_{LLC})^{\ell-1} \cdot m_{LLC}^{cold}(ROB) \cdot f(\ell) \tag{4.1}$$

Conflict misses lead to MLP in a similar way. However, we do not know how many loads in the ROB will cause a conflict miss. Therefore, we assume that conflict misses are uniformly distributed, and we estimate the number of conflict misses per ROB as follows: $M_{LLC}^{cf} \cdot \bar{L}(ROB)$. Following the same reasoning as for the cold-miss MLP, we estimate the conflict-miss MLP as follows:

$$MLP^{cf} = \sum_{\forall \ell} (1 - M_{LLC})^{\ell-1} \cdot M_{LLC}^{cf} \cdot \bar{L}(ROB) \cdot f(\ell) \tag{4.2}$$

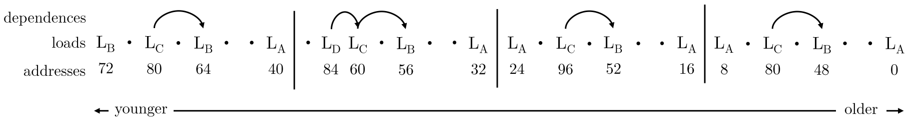


Figure 4.6: Illustrative virtual memory access stream.

Averaging Equations 4.1 and 4.2 based on the relative number of cold and conflict misses gives us an estimation for the overall MLP:

$$MLP = \frac{m_{LLC}^{cf}}{m_{LLC}} \cdot MLP^{cf} + \frac{m_{LLC}^{cold}}{m_{LLC}} \cdot MLP^{cold} \tag{4.3}$$

4.5 Stride-MLP Model

Since there are few cold misses present when analyzing a complete application, estimating burstiness is harder and the cold-miss MLP model is less accurate. The stride MLP model attempts to solve this and relies on a number of statistics that we capture on a per micro-trace basis. A micro-trace is defined as a small trace of instructions, e.g., 1000 instructions, extracted from the dynamic instruction stream. This approach is similar to the sampled simulation technique used in SMARTS [73]. The reason for considering micro-traces is to reduce profiling time, and more importantly, to be able to capture MLP burstiness. An average profile across a number of micro-traces would average out the statistics which would compromise model accuracy. See Chapters 5 and 6 for more details about sampling micro-traces and their accuracy.

Within each micro-trace, we measure a load-spacing distribution, inter-load dependence distribution, reuse distance distribution and stride distribution. Figure 4.6 serves as an illustrative example: it shows a trace of 32 instructions consisting of 16 loads with the oldest instruction appearing on the right. Loads are indicated as L_x with x indicating recurrences of the same *static* load instruction. Dependences between loads are shown through arrows; the addresses accessed are shown below the loads. We collect these distributions for each static load in each micro-trace.[3]

The load spacing distribution records a load's first position in the micro-trace along with the number of instructions in-between recurrences of the same static load. For load L_C in Figure 4.6, the load spacing distribution equals '$5; (8, 3)$' meaning that the first occurrence appears at position 5 and there are 8 instructions between the next three recurrences. The rationale behind the load spacing distribution is to capture the burstiness of loads, i.e., load instructions that miss in the on-chip caches and that occur within the same ROB is a necessary condition to expose MLP.

The inter-load dependence distribution quantifies inter-load data dependences in a statistical way. This is essentially the same distribution as measured

[3]Collecting and storing distributions requires, on average, 25× less disk space compared to storing all the instructions of a micro-trace.

for the cold-miss MLP. Inter-load dependences have an important impact on MLP, i.e., loads that depend upon each other (either directly or indirectly) cannot be issued simultaneously, hence they cannot expose MLP. The inter-load dependence distribution quantifies the probability that a load depends on any of the n previous loads in the instruction stream. For example, in Figure 4.6, load L_C (always) depends on load L_B. Because of this dependence, even if both loads L_B and L_C generate LLC misses, they will serialize their execution, and hence no MLP can be exploited.

The reuse distance distribution quantifies temporal locality by quantifying the number of (not necessarily unique) memory accesses between two accesses to the same memory location. This reuse distribution is then transformed using StatStack [28] into a stack distance distribution, which quantifies the number of *unique* accesses between two accesses to the same memory location. Note that the measured reuse distance distribution is different from the one StatStack normally uses. Rather than sampling throughout the complete application, all accesses that are sampled originate from the same micro-trace.

Once the stack distance distribution is known, it is trivial to derive the miss rate assuming a fully associative LRU cache of arbitrary size, i.e., if there are more unique accesses between two accesses to the same memory address than there are sets in the cache, the last access to the same memory address will be a miss. Note also that the reuse distance distribution is measured per static load, hence it enables estimating the miss rate per static load for any cache size. Moreover, we can use the reuse distance distribution for predicting hits and misses at all levels of cache, from the L1 cache to the LLC.

The last distribution we consider is the stride distribution. A stride is defined as the relative memory address difference between two subsequent recurrences of the same static load. The stride distribution collects this stride information. Whereas the reuse distance distribution quantifies temporal locality in a statistical way, the stride distribution is a measure for spatial locality. The stride distribution is also critical to model stride-based prefetching, as we will describe in Section 4.9.

Memory accesses do not always follow a neat stride pattern, i.e., some patterns can be a mixture of several strides, other memory accesses may appear to be random. We classify loads into three categories based on their access patterns. The first category includes loads that follow *some* stride pattern. The second category includes loads that occur only once in our micro-trace. The third category includes loads that do not fit in either of the above two categories; we refer to this category as random-strided loads.

For the strided-load category, we search for up to four distinct strides per load, and we use a cutoff percentage to filter out accesses that are not part of a real stride pattern. To categorize a load as an instruction with a single stride, one element in the stride distribution needs to occur at least 60%. For a two-strided load, their cumulative percentage needs to exceed 70%, for a three-strided load 80%, and for a four-strided load 90%. We always choose the simplest stride pattern; this means that if the cumulative percentage of

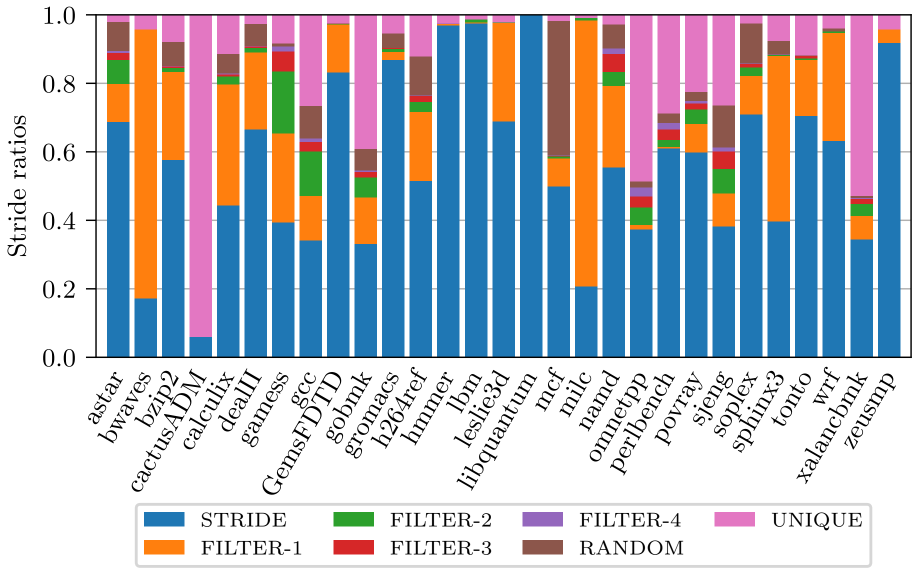


Figure 4.7: Ratio of all stride categories for the SPEC CPU 2006 benchmarks.

occurrence exceeds a threshold, we stop searching for additional strides, such that we can easily filter out random strides.

For example, in Figure 4.6, L_A recurs six times and exhibits a single-strided pattern with stride 8. Load L_B recurs five times with memory addresses: $48, 52, 56, 64, 72$. There are two strides of 4 and two strides of 8. Each stride thus has an occurrence equal to 50%, hence this load is classified as a two-strided load.

Figure 4.7 shows the relative occurrence of the specific stride categories for all SPEC CPU 2006 benchmarks. The category with label 'STRIDE' is used for loads where the profiled stride distribution features exactly one stride. The 'FILTER-1' through 'FILTER-4' categories are loads for which the filter algorithm was applied to categorize them using the specified cutoff percentages. The unique and random bars are also constructed following the above definitions. Note that 'STRIDE' and 'FILTER-1' categories both contain loads exhibiting one stride and are treated as such in the model, but that we split them in Figure 4.7 to highlight the fact that a significant number of the one-strided loads exhibit exactly one stride which does not require filtering.

We can observe that, while for a majority of the benchmarks, most loads fall in the category with exactly one offset, we do need the filtering algorithm to categorize a meaningful amount of loads as either strided or random-strided loads. There are three exceptional benchmarks, *cactusADM*, *omnetpp* and *xalancbmk*, for which more than 50% of the loads are unique loads. One possible explanation for this is that these benchmarks have large loops (or unrolled loops) causing the micro-traces to overlap with only one iteration and thus not finding a real stride pattern. However, this is not a problem since we can still rely on StatStack to predict whether that unique load will miss or not.

The distributions as just described need to be collected only once per application, from which we can predict MLP for a range of architecture configurations. We first generate a virtual instruction stream from these distributions; this virtual instruction stream is built up as a data structure by the MLP modeling software. We then hover over this virtual instruction stream with an abstract MLP model to estimate the amount of MLP for a particular architecture. This is done for each micro-trace.

The load spacing distribution is first used to build up a skeleton virtual instruction stream. We position loads in the instruction stream using the load spacing distributions which determine the first position of each static load in the stream as well as the subsequent recurrences of the load; this is done for all static loads in the micro-trace. We then use the stride distribution to assign (relative) memory addresses for each load occurrence of the same static load. The stride distribution points out hits and misses in the cache, at least for those loads that exhibit a strided access pattern. We predict hits and misses at all levels in the cache hierarchy. More in particular, we mark the first access of a stride pattern as a miss and we mark the following accesses that fit the same cache line as hits. We use the reuse distance distribution and StatStack to predict whether an address has been used before and the respective load will turn into a hit or a miss. We leverage the inter-load dependence distribution to impose dependences between loads.

The abstract MLP model then hovers over this virtual instruction stream to estimate MLP for a particular architecture with a specific ROB size. MLP is defined as the number of outstanding memory requests (LLC misses) if at least one is outstanding. The abstract model breaks up the virtual instruction stream into ROB-sized instruction sequences over which it estimates the available MLP. We considered two possibilities: an ROB that slides versus steps over the instruction stream; both gave similar results according to our preliminary results, hence we opt for the stepping approach which is slightly simpler to implement and less compute intensive. For a given ROB-size sequence of instructions, MLP is computed as the number of independent main memory accesses in the ROB. MLP for the micro-trace is computed as the average MLP across all ROB-sized instruction sequences.

4.6 Modeling MSHRs

The MLP models discussed so far make a number of simplifying assumptions. It assumes that all independent memory references access main memory and return their data simultaneously. In addition, it does not consider hardware prefetching. The next sections discuss extensions to the MLP model to overcome these assumptions.

Modern processors typically feature Miss Status Handling Registers (MSHR) to coalesce multiple requests to the same cache line. An MSHR entry is allocated upon an access to a cache line that is not yet outstanding. Subsequent requests to an already outstanding cache line are then coalesced, avoiding yet another request being sent to the next level in the memory hierarchy. The size of the Miss Status Handling Register (MSHR) is (obviously) limited, and hence it may limit MLP, i.e., a memory access to a not yet outstanding cache line may be stalled if the MSHR runs out of available entries.

In this work we consider an MSHR table at the L1 data cache level, however, the approach can be trivially generalized to MSHRs at other levels of cache. We predict whether the number of outstanding L1 data cache misses in the instruction stream exceeds the number of MSHR entries. If it does, we compute a scaling factor that accounts for the extra latency added to the loads waiting for an available MSHR entry. Note that his model can work for both MLP models. This model differs from the one proposed by Chen et al. [19] in which the MLP is simply capped to an upper bound; our model puts a 'soft' cap on the MLP and models partially overlapping memory accesses.

We estimate the impact of a limited number of MSHR entries as follows. If we use the cold-miss MLP model, we calculate the number of L1 misses per ROB using a uniform spread of the number of L1 misses in a complete ROB. If we use the stride MLP model, we split the micro-trace into ROB-size sequences of instructions of which the first instruction is a (predicted) access to main memory and the last instruction the one that still fits within the ROB. The first few memory accesses that miss in L1 all fit in the MSHR table and are hence considered to execute in parallel. All subsequent main memory accesses that would overflow the MSHR table have to wait until one of the outstanding accesses is resolved. Hence, they only partially overlap with the previous accesses. We model this phenomenon by considering the time it has to wait for a free MSHR slot. Intuitively, this means that the first part of the latency is serialized and the remaining part is hidden underneath another access. This results in the following equation which puts a 'soft' cap on the exploitable MLP:

$$MLP = DRAM_{MSHR} + DRAM_{wait} \cdot \frac{T_{DRAM} - T_{MSHRfree}}{T_{DRAM}} \tag{4.4}$$

with $DRAM_{MSHR}$ the number of main memory accesses in the MSHR table, i.e., this is the number of parallel main memory accesses; $DRAM_{wait}$ is the number of main memory accesses that have to wait; T_{DRAM} equals the main memory access latency and $T_{MSHRfree}$ is the average time before an MSHR slot becomes available, which is computed as the weighted average access latency across all allocated MSHR entries.

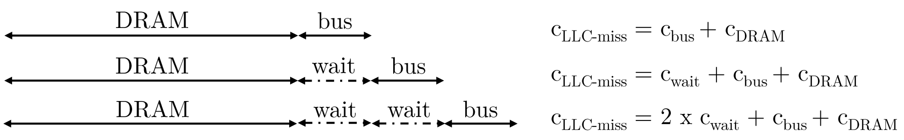

DRAM
bus
DRAM
wait
bus
DRAM
wait
wait
bus
c_LLC-miss = c_bus + c_DRAM
c_LLC-miss = c_wait + c_bus + c_DRAM
c_LLC-miss = 2 x c_wait + c_bus + c_DRAM

Figure 4.8: Visualization of the queuing delay due to multiple concurrent memory accesses.

4.7 Main Memory Bus

We find that for some applications, due to their bursty memory behavior, the available memory bandwidth is often not sufficient, resulting in memory controller congestion and queuing delays. To model this, we assume that the number of concurrent misses equals the MLP on average. Therefore, the first miss has a bus latency equal to the bus transfer time, i.e., the size of a cache block divided by the width of the memory bus. The second concurrent miss has to wait until the first miss releases the bus, so its bus latency equals twice the bus transfer time. And the third miss has a bus latency of three times the bus transfer time, etc. This is visualized in Figure 4.8. Note that this is only valid if the processor features only one memory channel. However, generalizing this to multiple DRAM channels could be done by assuming a uniform distribution of the accesses over the different channels.

The average bus latency for MLP' concurrent accesses (we define MLP' next) therefore equals:

$$c_{bus}(MLP') = \frac{1}{MLP'} \sum_{i=1}^{MLP'} i \cdot c_{transfer} = \frac{MLP'+1}{2} c_{transfer} \quad (4.5)$$

We use linear interpolation to deal with non-integer MLP numbers.

The MLP factor only takes into account loads that miss in the LLC, because store misses usually do not incur a penalty for the core performance (except when they prevent other loads to issue because of a structural constraint, e.g., a full write buffer or an exhaustion of MSHRs). However, they do need to access memory, so they have an impact on memory bandwidth contention. In fact, we find that for benchmarks that have a lot of LLC store misses, memory bandwidth contention is underestimated. We compensate for this by rescaling the MLP to include the store misses:

$$MLP' = MLP \cdot \frac{m_{LLC}^{load} + m_{LLC}^{store}}{m_{LLC}^{load}} \quad (4.6)$$

where m_{LLC}^{load} and m_{LLC}^{store} are the number of LLC load and store misses, respectively. Note that all previous LLC miss counts only include load misses. This MLP' is used in Equation 4.5 to calculate the average bus transfer time.

4.8 Chained LLC Hits

The last term in Equation 3.1 is the penalty of LLC hits, i.e., loads that miss in the L1 and L2 caches, but hit in the LLC.

One of the most important features of a superscalar out-of-order processor is its ability to hide instructions with short latency, e.g., floating-point operations or loads that hit in one of the higher (L1 and L2) cache levels. Interval analysis assumes that the latency of an operation can be hidden if that latency is smaller than the time to fill up the ROB, i.e., the ROB size divided by the dispatch width. In our configuration, the only latency that is larger than the ROB fill time is the main memory access time due to a LLC miss. However, the hit latency of the LLC is for most configurations close to this threshold (e.g., 30 cycles LLC hit latency, and an ROB of 128 and dispatch width of 4, which results in 32 cycles fill time). We find that when two or more LLC hits depend on each other, we do notice some penalty. We call this the *chained LLC hit penalty.*

An example of this problem is shown in Figure 4.9, which is a visualization of one billion instructions of the `gcc` benchmark executed on our reference architecture as simulated by Sniper and calculated by our model with and without modeling chained LLC hits. The first 400 million instructions are executed at a CPI of around 0.8, followed by a few peaks due to many DRAM accesses. The interesting region however starts around 650M instructions with the average CPI rising to around 3. The reason for this is in part an increase in the number of branch misses, but also, and more importantly, a substantial increase in the number of LLC hits, which leads to a high probability of multiple dependent LLC hits. The LLC-chaining component contributes around 20% to the total CPI (see the delta between the 'model' and 'model, no LLC chaining' curves in Figure 4.9). Not including the LLC hit chaining term, the estimation error on the total execution time for `gcc` equals -12.3%, while with this component, the error is reduced to -3.6%. Note that the underestimation in performance mainly originates from the overestimation of the MLP at around 500M instructions, rather than from the error on the LLC chain penalty.

Our goal is to estimate the penalty of chained LLC hits without involving additional profiling. To estimate the penalty due to chained LLC hits, we first calculate the average number of LLC load hits in one ROB, $h_{LLC}(ROB)$, as the LLC hit rate (as estimated by StatStack) times the average number of loads in the ROB. Contrary to MLP calculation, where we want to calculate the number of independent LLC misses, we now need to compute the number of LLC hits that are on the same dependence path. All loads on a dependence path will be executed sequentially because of the dependences between them, so all LLC hits on a path will be serialized. To find this number, we reuse the load dependence distribution that is profiled for MLP calculation. All loads that are first on a path (i.e., loads that are independent of all other loads) initiate a new possible path with dependent LLC hits. So the number of independent loads equals the number of dependence paths with loads on it, denoted $p_{load}(ROB)$. Assuming that LLC hits are uniformly distributed across

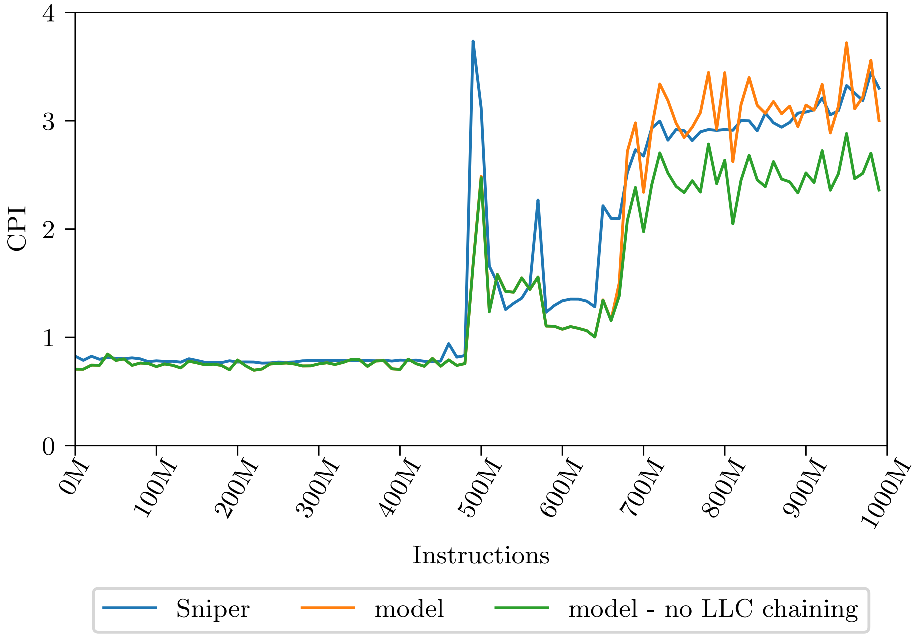


Figure 4.9: CPI variation over time for `gcc` with and without the LLC hit chaining component compared to Sniper.

the dependence paths, the average number of LLC hits on a path (LLC hit chain or LHC) can be estimated as follows:

$$LHC_{avg} = \frac{h_{LLC}(ROB)}{p_{load}(ROB)} \tag{4.7}$$

However, the LLC hit chain penalty is not determined by the average chain of LLC hits, but by the longest chain. The longest chain is at least as long as the average chain, and is bounded by the number of LLC hits in the ROB, $h_{LLC}(ROB)$, as well as by the largest number of loads on a dependence path. We cannot deduce the latter from the load dependence distribution, so we approximate it by the average number of loads on a path, $\overline{lop}(ROB)$. The maximum number of LLC hits on a path thus equals:

$$LHC_{max} = \min(h_{LLC}(ROB), \overline{lop}(ROB)) \tag{4.8}$$

To calculate the expected value of the largest number of LLC hits on a dependence path, we assume that we have at least LHC_{avg}, and that the remaining LLC hits that can possibly belong to this path, i.e., $LHC_{max} - LHC_{avg}$, are distributed uniformly across all $p_{load}(ROB)$ paths. The expected longest chain of LLC hits therefore equals:

$$LHC_{exp} = LHC_{avg} + \frac{LHC_{max} - LHC_{avg}}{p_{load}(ROB)} \tag{4.9}$$

The resulting penalty then equals the chain length times the LLC hit latency c_{LLC}:

$$P'_{hLLC}(ROB) = c_{LLC} LHC_{exp} \tag{4.10}$$

As explained before, latencies that are smaller than the ROB fill time are hidden by out-of-order execution. Hence, we have to subtract the cycles it takes to fill the ROB from the penalty calculated in Equation 4.10, which yields the average penalty for a window of ROB instructions:

$$P_{hLLC}(ROB) = \max\left(0, P'_{hLLC}(ROB) - \frac{ROB}{D_{\mathit{eff}}}\right) \tag{4.11}$$

The total penalty for the full application thus equals this penalty times the number of windows of ROB instructions across the entire instruction stream:

$$P_{hLLC} = P_{hLLC}(ROB) \cdot \frac{N}{ROB} \tag{4.12}$$

4.9 Hardware Prefetching

In most contemporary processors there are one or more prefetchers present. The goal of a prefetcher is to predict which data will be requested by load instructions in the future. If it can successfully predict this, it can request that data before the load instruction is executed. In doing so, the data for that load will already be in one of the cache levels, therefore significantly shortening the latency of said load instruction. There exist a numerous amount of work on prefetching going from stride prefetching [33] to prefetching using a global history buffer [51] and even prefetchers that are trained using machine learning techniques [55].

A key feature of the stride MLP model is that it enables estimating the performance impact of stride-based prefetching. In this work, we consider a stride prefetcher that tracks the stride patterns of a number of static loads (per-PC stride prefetching) [33].

Figure 4.10 shows a stream of memory accesses and the associated table indicating the access pattern for static loads A through D. A, B and D are loads exhibiting a fixed stride of 16, 128 and 8k, respectively, while C exhibits a random access pattern. Depending on the exhibited stride pattern and the location of the loads in the dynamic instruction stream they can be prefetchable or not as detailed below.

A stride prefetcher needs to keep track of previously executed loads and their addresses to compute a load's stride pattern. There is obviously a limit to the number of static loads the prefetcher is able to track. If the number of static loads occurring between two recurrences of the same static load is bigger than the maximum tractable loads in the prefetch table, the recurring load cannot be used to prefetch the future occurrences.

A_1	-	B_1	-	A_2	-	D_1	C_1	A_3	B_2	-	-	B_3	-	C_2	D_2	A_4	B_4	-	C_3	D_3	-	D_4	C_4
32	-	256	-	48	-	4k	8	64	384	-	-	512	-	140	12k	80	640	-	150	20k	-	28k	104

Static load	Past address	Current address	Future addresses
A	32	48	64 - 80
B	256	384	512 - 640
C	8	140	150 - 104
D	4k	12k	20k - 28k

Figure 4.10: Example of the different effects to take into account when modeling a stride prefetcher (k is shorthand for a multiplier of 1024).

To illustrate this, consider the example instruction stream in Figure 4.10, but with a prefetcher table counting only two entries. In that case, the occurrences of static load A, A_3 and A_4, are predictable. After all, when A_2 is executed, A_1 is still present in the prefetcher table, so we can calculate its stride, 16, and predict that the next addresses will be 64 and 80. However, the occurrences D_3 and D_4 of static load D are not prefetchable because by the time D_2 is encountered, D_1 has been removed from the prefetcher table which now contains B_3 and C_2.

For our prefetcher model, we can emulate the limited size of the prefetcher table by walking over the virtual instruction stream generated for the stride MLP model and keeping a limited-size list tracking only the last x static loads. For each load we observe in the virtual instruction stream, we can then check this list. If a load is executed for which the past occurrence is still part of the list, we can mark the future occurrences of that load as prefetchable. If the executed load is not part of the list anymore, we mark it as non-prefetchable.

Second, prefetchers often only prefetch within a DRAM page, meaning that if two subsequent accesses are not part of the same virtual memory page, the second one will not be prefetched. For example, load D in Figure 4.10 exhibits a stride of 8k or 8×1024 while a DRAM page is often only 4096 bytes big. Thus, even though it would be possible to prefetch D_3 based on the occurrences of D_1 and D_2, it is not prefetched. We include this behavior in the prefetcher model by considering the stride between two subsequent accesses by the same load. If the stride extracted from the stride distribution exceeds the size of a DRAM page, we mark each subsequent occurrence of that static load as non-prefetchable.

The third effect that requires modeling relates to timeliness. If the prefetcher starts fetching new data just before the data is requested, the prefetch will not be timely, and the latency of the load can only be hidden partially. For example, load A exhibits a stride of 16 and load A_3 and load A_4 occur in different ROBs far enough from each other. Thus, the prefetcher can fetch the data timely as the ROB will block on the first two A-accesses that miss. However, consider loads B_2 and B_3, which occur close to each other as an example. Even though load B exhibits a perfectly regular stride pattern, prefetching the data needed by B_3 cannot happen in time because it is needed almost immediately after B_2 is executed and main memory is very slow compared to the processor's execution speed.

In our prefetcher model we model timeliness by assuming that a prefetch for a load that is ROB-size instructions away in the dynamic instruction stream, is timely. If the load appears in the same ROB-size instruction window as the prefetch, we only subtract a fraction of the latency equal to the time it would take for the latter load to hit and stall the ROB head. This is shown in Equation 4.13.

$$c_{prefetch} = \begin{cases} 0 & , I_{L2} - I_{L1} \geq ROB \\ c_{DRAM} - \frac{(I_{L2}-I_{L1})}{D_{eff}} & , I_{L2} - I_{L1} < ROB \end{cases} \tag{4.13}$$

Here, I_{L1} and I_{L2} indicate the positions of the two loads in the dynamic instruction stream.

Note that one downside of the sampling methodology we employ throughout this work is that we cannot accurately predict whether a prefetch delivers useful data and thus also cannot predict whether the prefetcher pollutes the cache.

4.10 Memory Power Modeling

Generating power predictions using McPAT also requires supplying the micro-architecture parameters related to the memory hierarchy. For each cache level, its size, associativity, block size, and latency are supplied. This allows McPAT to build the core floorplan and estimate the static power consumption of the cache hierarchy.

Next to describing the cache hierarchy, we also supply the respective activity factors to estimate the dynamic power consumption. Most of these are readily available from the predictions by StatStack, such as the number of misses at each cache level. Note that, compared to the performance model, the power model requires extra inputs, such as the number of store misses and writebacks, because these do impact power consumption. These are not needed for the performance model, because there, it is assumed that they have no impact. Store misses are predicted by StatStack in a similar fashion as load misses which we explained in Section 4.2. The number of writebacks is more difficult to estimate, but we noticed that they have a very small impact on total power consumption, so we omit them from the power model.

Chapter 5

Sampling Methodology

Profiling an application to collect all the data to model the core takes a significant amount of time. To speed up the profiling step, we employ aggressive sampling methods. This is similar to adopting sampling in simulation as described in Section 2.2. While these sampling methods have the potential to speed up the profiling step significantly, they also introduce errors. In the following sections, we explain the sampling methods and quantify their error.

5.1 Instruction Mix Sampling

To collect statistics related to the instruction mix, we profile a small amount of instructions called a *micro-trace* and then fast forward through the dynamic instructions stream for a large amount of instructions. We call the combination of the micro-trace and the fast forwarded instructions a *window*. The ratio of the number of instructions in a micro-trace on the number of instructions in a window is called the *sample rate*. This sampling approach is similar to the technique employed in SMARTS [73]. Figure 5.1 visualizes this method. For example, to generate the instruction mix histograms and dependence chain statistics, we profile a micro-trace of 1000 instructions and then disable profiling until the end of the window is reached at instruction 1 million. We motivate the choice for these sampling parameters in Section 6.2.

Figure 5.2 shows a comparison of the instruction mix of SPEC CPU 2006 workloads with sampling enabled and disabled. The left bar shows the measured instruction mix when profiling micro-traces of 1000 instructions every 1 million instructions while the right bar shows the profiled instruction mix without sampling. Note that we show the instructions broken down into their respective micro-operations. One can visually verify that the error caused by sampling the instruction mix is small. We also quantify the error by verifying the sampling error for each instruction category separately using Equation 5.1. For each instruction category we subtract the extrapolated, sampled amount

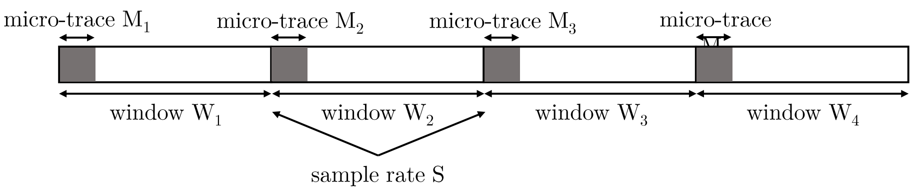


Figure 5.1: Visualization of the micro-traces and windows in an instruction stream.

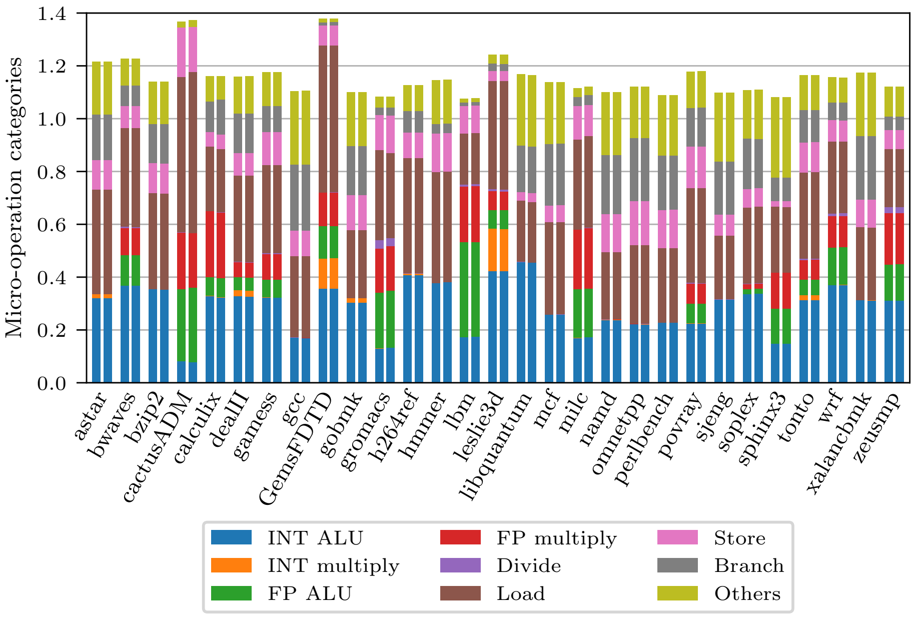


Figure 5.2: Comparison of the sampled (left bar) and non-sampled instruction mix (right bar) using 1 / 1000 sample rate.

from the non-sampled amount and divide by the total number of instructions. Note that this denominator is used to mitigate skewing of the average error by instruction categories with low frequencies.

$$\forall_{category\ c},\ error_c = \frac{\#micro\text{-}ops_{c,sampled} - \#micro\text{-}ops_{c,not\ sampled}}{\sum_c micro\text{-}ops_c} \tag{5.1}$$

This equation shows that the average error for the instruction categories is 0.08%, with the maximum error being 1.8%. Thus we can confirm that the prediction error caused by sampling the instruction mix should be small.

5.2 Dependence Chain Interpolation

The algorithm described in Section 3.3 calculates the dependence chains for one ROB size at a time. Since the profiling step for our analytical model needs to be micro-architecturally independent, we calculate the dependence chains for arbitrary ROB sizes. We take a fixed set of ROB sizes and interpolate within that set to for different ROB sizes. Our default set of ROB sizes ranges from 16 to 256 entries. Within this set we profile every ROB with a size of a multiple of 16. We do not calculate the dependence chains for all ROB sizes primarily because only a few of them are of actual interest when developing a processor (e.g., a ROB size of 87 is not really interesting). Compared to calculating dependence chains for all ROBs, this method yields a speedup of 8×.

However, the sampling method introduces two problems. According to the last line in Algorithm 3.2, any ROB size can be required to calculate the branch resolution time. Furthermore, when optimizing a processor, it is possible that the ROB of interest was not profiled and it is undesirable to re-profile an application solely for that reason. Thus, we interpolate the dependence chains for the chosen set of profiled ROB sizes to other ROB sizes. Based upon the observation of Eyerman et al. [32], we can approximate the lengths of dependence chains for non-profiled ROB sizes using a logarithmic fit following Equation 5.2.

$$chain\, length = a \cdot log(ROB) + b \tag{5.2}$$

Figure 5.3 shows an example for the *astar* benchmark to show that such a logarithmic fit works well. The dotted lines visualizing the fitted dependence chains only deviate from the full lines, which are the measured dependence chains, for very small ROB sizes ($\leqslant 16$). These sizes are of no interest for an out-of-order processor, but we do need the size of the dependence chains for these ROB sizes to predict the branch resolution time as described in Section 3.5.

Parameters a and b are calculated using the least square method following Equation 5.3 and 5.4 where x is the ROB size and y the length of the dependence chains. We calculate a fit between each pair of profiled ROBs separately (i.e., between 16 and 32, 32 and 48, etc.) as this results in a smaller error than an overall fit using all points.

$$b = \frac{N \cdot \sum log(x) \cdot y - \sum log(x) \cdot \sum y}{N \sum log(x)^2 - (\sum log(x))^2} \tag{5.3}$$

$$a = \frac{\sum y - b \cdot \sum log(x)}{N} \tag{5.4}$$

The average error for each SPEC CPU 2006 benchmark is shown in Figure 5.4. The approach of fitting yields minimal errors with even the biggest error being smaller than 1%. The average error for all SPEC CPU 2006 benchmarks is 0.34%, 0.23% and 0.61% for the average path, average branch path and critical path, respectively.

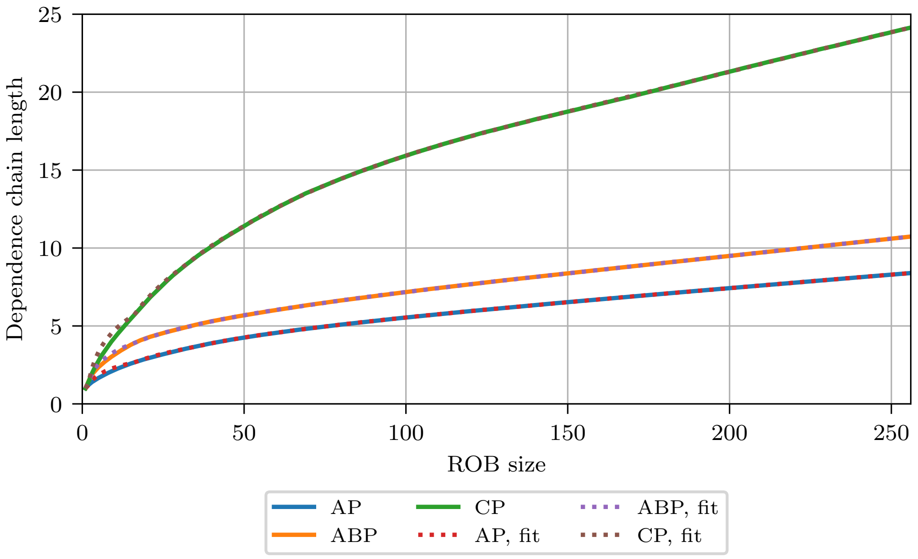


Figure 5.3: Comparison between profiled and interpolated AP, ABP and CP dependence chains for *astar*.

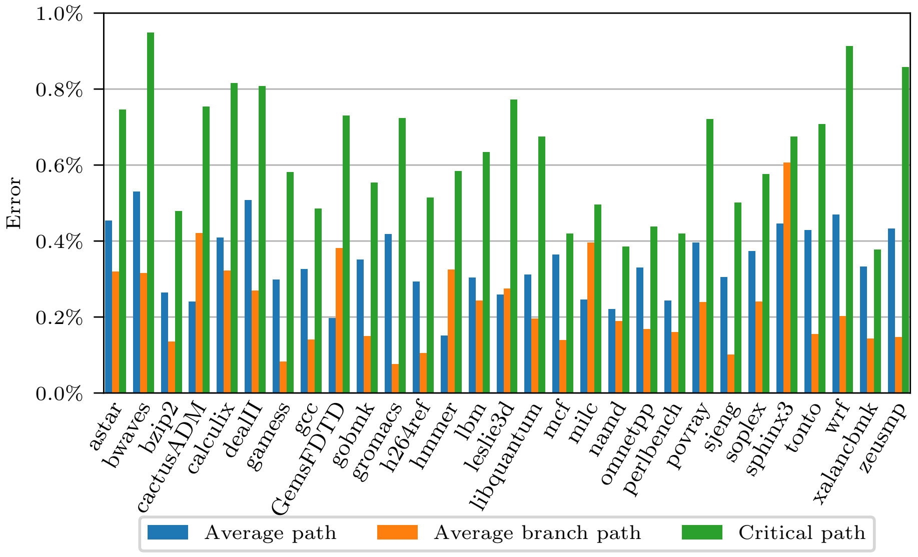


Figure 5.4: Error on the dependence chain lengths due to interpolation between ROB sizes for all SPEC CPU 2006 benchmarks.

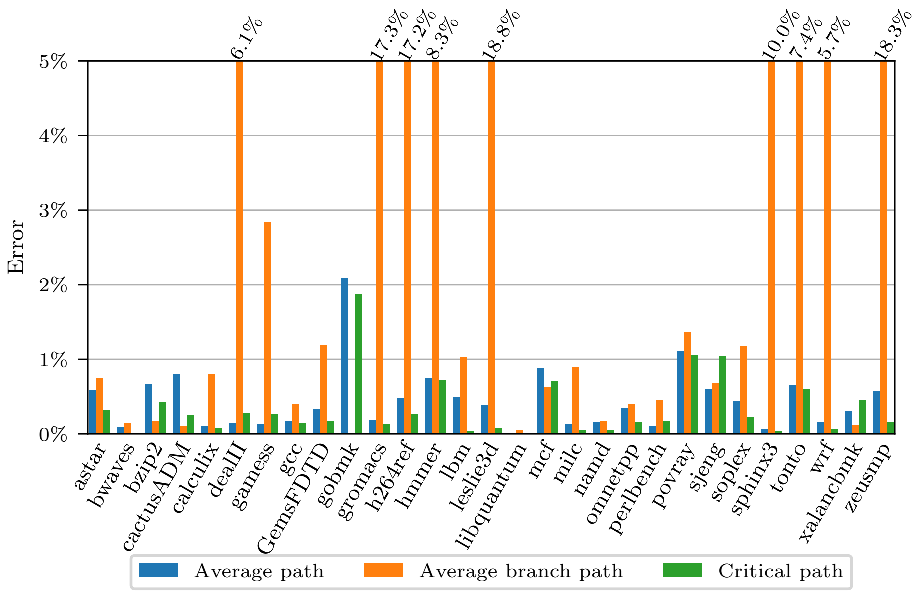


Figure 5.5: Error on the dependence chain lengths due to micro-trace sampling for all SPEC CPU 2006 benchmarks.

5.3 Dependence Chain Sampling

While sampling the instruction mix already leads to some speedup, an appropriate sampling method for computing dependence chains is obviously more important. Prior work by Genbrugge et al. [34] optimized the calculation of dependence chains by calculating them once for each ROB-sized interval of instructions throughout the complete instruction stream. However, because we need the dependence chains for a range of ROB sizes, this method yields little speedup.

We chose to follow the same method of analyzing micro-traces of 1000 instructions every one million instructions. Algorithm 3.1, described in Section 3.3, lends itself well for sampling. Instead of feeding the complete instruction stream to the algorithm, only the micro-trace is passed to it. Given the $O(N \cdot B)$ algorithm complexity, this leads to a significant speedup.

We quantify the sampling error in Figure 5.5. The errors on the average path and critical path are negligibly small at 0.45% and 0.34%, respectively. However, the average error for the average branch path is 4.22%. Eight benchmarks exhibit a sampling error larger than 5%. The explanation for these outliers is that not every micro-trace contains a lot of branch instructions which can thus lead to a larger sampling error. Taking larger traces, e.g., 10k instructions, improves the average sampling error for the average branch path from 4.22% to 2.81%, but also slows down the profiling by almost 10×.

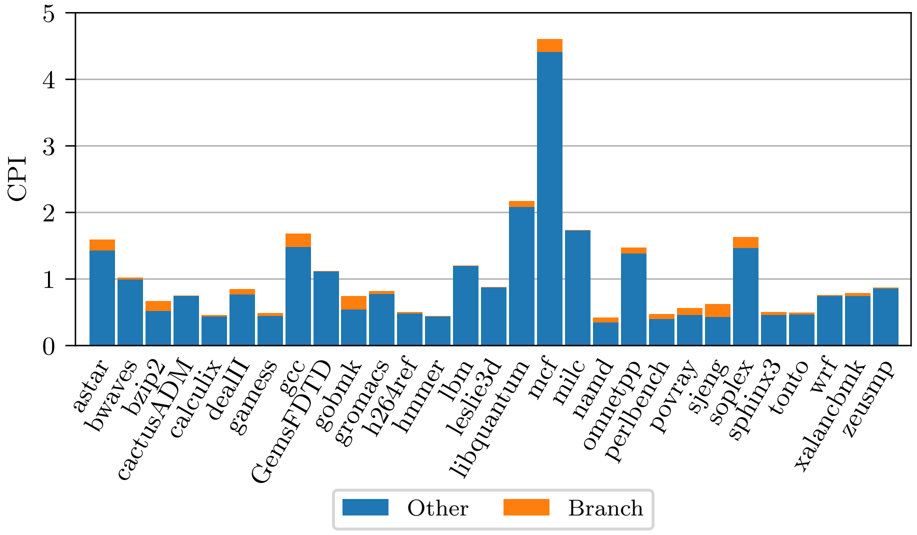


Figure 5.6: Relative contribution of the branch component to the total execution time for one billion representative instructions from the SPEC CPU 2006 benchmarks.

Note that the ABP metric is only used for calculating the branch resolution time, which is only one part of the branch penalty. The branch misprediction component does not necessarily influence the total execution time significantly. Figure 5.6 confirms this by visualizing the relative contribution of the branch component to the total execution time for all SPEC CPU 2006 benchmarks using Sniper. Since the branch component is relatively unimportant, partially due to the low branch misprediction rates, we favor faster profiling and deem the sampling error acceptable.

5.4 Memory Sampling

5.4.1 Cache Miss Rates

Measuring reuse distances for all memory addresses in the dynamic instruction stream would be very time-consuming. Therefore, StatStack reduces this overhead through sampling. It splits the stream of memory operations into different, subsequent intervals of memory accesses called *bursts*. By default, these bursts consist of 600.000 memory accesses. Within these bursts, StatStack samples and tracks the reuse of one in one thousand memory addresses. Thus, per burst, it tracks 600 memory accesses. Since this approach yields good accuracy in the original work, we do not modify it here. Note that these bursts can be outlined differently compared to the instruction windows we use to collect the other statistics. Hence, we need to interpolate different bursts to correlate them to the instruction windows.

5.4.2 Memory-Level Parallelism

In Sections 4.4 and 4.5 we described two techniques to model memory-level parallelism. The cold-miss MLP model relies on cold-miss distributions, while the stride-MLP model relies on stride and reuse distance distribution. The key take-away is that it is impossible to sample the memory address stream, and still arrive at an accurate cold-miss distribution, while it is possible to sample stride and reuse distance distributions. Thus, collecting the distributions for the stride-MLP is around 40% faster than collecting cold-miss distribution.

The sampling approach to collect stride and reuse distributions is the same as collecting instruction mix and dependence information. We profile all load instructions in a micro-trace of one thousand instructions every one million instructions. This is not only necessary to speed up the profiling phase, but also to achieve good prediction accuracy for the MLP since averaging out statistics over large traces would comprise MLP-burstiness estimations.

Note that sampling using a micro-trace approach implies we had to modify the default StatStack sampling approach. Instead of sampling uniformly over a burst of memory accesses, we sample every access in the beginning of the burst and then track its reuses. We also adapted StatStack to calculate miss rates per static load instead of for all loads simultaneously. The reason for using this approach is that we need to correlate stride distributions with the miss rates as predicted by StatStack, which would otherwise be impossible.

Chapter 6

Evaluation

6.1 Experimental Setup

The main goal of this work is to develop a mechanistic model that can predict performance and power accurately and faster than simulation. Our profiling tool uses Intel's Pin [48], a binary instrumentation tool for collecting application characteristics and store them in a binary file format using Google's Protobuf [4]. The evaluation of the model is performed using a Python framework [8].

We use two different evaluation baselines. For evaluating performance and power prediction accuracy of the complete model as described in Sections 6.2 through 6.5, we use the 29 SPEC CPU 2006 benchmarks with reference inputs. Because we have to simulate all of the designs to determine the accuracy of the model, we created a 1 billion instruction SimPoint [59] for each benchmark. As explained in Chapter 2, SimPoints are a way of creating a representative sample of instructions for a complete application. We use the Pinball technology [54] to create checkpoints at the beginning of the SimPoint. All results in Sections 6.2 through 6.5 are generated using the cold-miss MLP technique.

For evaluating the stride-MLP technique in Section 6.6, we use the train inputs of the SPEC CPU 2006 benchmark suite and we run the benchmarks to completion. Using the reference inputs would be infeasible as it would take multiple months of simulation time. We use a periodic sampling strategy to limit simulation time while still covering the entire benchmark execution. To compute the ground truth to evaluate the model against, we fast-forward 800M instructions, warm up the memory hierarchy for 100M instructions, and then simulate 100M instructions in detailed mode; this is repeated till the end of the execution. We consider a similar sampling strategy for collecting our profile: we fast-forward 800M instructions, enable StatStack for the next 100M and collect our complete profile during the next 100M instructions; this procedure guarantees that the profile corresponds to the detailed simulation region. Note

Parameter	Value
Dispatch width	4 wide
Instruction queue	43 entries
ROB entries	128 entries
Branch predictor	pentium-M predictor [65]
L1-I cache	32 KB, 4-way associative, latency 1 cycle
L1-D cache	32 KB, 8-way associative, latency 4 cycles
L1-MSHR	10 entries
L2 cache	256 KB, 8-way associative, latency 8 cycles
L3 cache	8 MB, 16-way associative, latency 30 cycles
Memory bandwidth	8 GB/s
Memory latency	120 cycles

Table 6.1: Core configuration for our reference architecture, based on the Intel Nehalem processor.

that StatStack's accuracy could be further improved by allowing for a cooldown phase.

The reason for using the second baseline is twofold. First and foremost, we want to cover the complete benchmark to make sure we model all different phases in the benchmarks rather than just one specific phase. Secondly, we need to eliminate cold misses as much as possible such that the estimation of MLP-burstiness is not influenced too much by cold misses. We verified that, by employing this technique, for our data-intensive benchmarks, the average number of cold misses is only 1% of the total amount of cache load misses. Note that we could also use the previously described SimPoints prepended by a cache warmup of billions of instructions, but we did not have the simulation infrastructure to achieve this.

Obtaining a ground truth with detailed simulations is achieved using the most accurate core model within Sniper, which has been validated against real hardware [17]. Power measurements are done using the McPAT-tool [46] included with Sniper for a 45nm chip technology.

6.2 Performance Prediction

6.2.1 Absolute Accuracy

We first evaluate the accuracy of our model against a Nehalem-based reference architecture as described in Table 6.1. Figure 6.1 shows the CPI of the benchmarks for the reference configuration obtained using our model (left bar) and through simulation with Sniper (right bar). The errors of the model versus Sniper are indicated on top of the bars. The average absolute error across all benchmarks equals 7.6%. There are positive and negative errors, which shows that our model is not biased. A maximum error of 22% is observed for *gromacs*, which is due to severe functional unit contention at very small

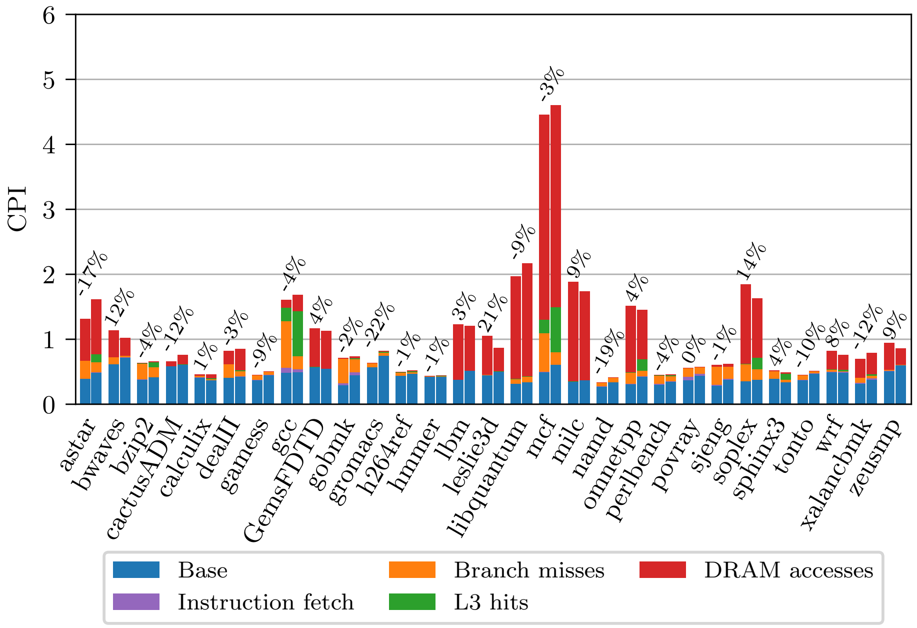


Figure 6.1: CPI stacks generated by the model (left bar) and by Sniper (right bar), and the error of the model versus Sniper simulations (top).

timescales. We do not model this very accurately because we use micro-traces of 1000 instructions, over which this fine-grained behavior can be averaged out.

Figure 6.1 breaks up the overall CPI into a number of components. The model, visualized in the left bar according to Equation 2.3, consists of an addition of five components reflecting different penalties. We can represent each component separately in a stack, such that the top of the stack equals total cycle count. By dividing the components by the number of instructions, we get the respective CPI stack components. More detail about CPI stacks is provided in Section 6.4. For the right bar, we use the built-in CPI stack generator of Sniper.

The CPI stacks generated by Sniper and by our model match well, which suggests that the overall model accuracy is not much embellished by compensating under- and overestimations. Note that some of the differences stem from the fact that there is no unambiguous way of defining CPI stacks in an out-of-order processor, because events can occur concurrently. Hence, whether cycles are accounted to one or another event may lead to small differences in how CPI stacks are constructed. The most noticeable example is an L3 cache hit that is part of the dependence path leading to a mispredicted branch: it is accounted to the L3 component in Sniper, because at the time the L3 cache hit occurs, it is impossible for Sniper to detect that it is part of a path to a mispredicted branch. Our model, on the other hand, accounts the miss latency to the branch miss penalty, because we model L3 hits as long-latency instructions that usually do not incur stalls. This is why the branch component for the model CPI

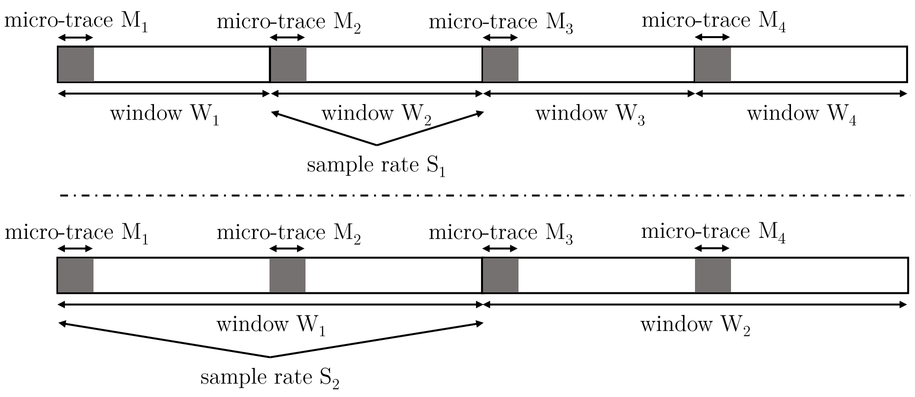


Figure 6.2: Visualization of the different sampling approaches to collect micro-traces within windows in an instruction stream.

stack tends to be larger than for the Sniper CPI stack, and vice versa for the L3 hit component, with *gcc* being the most notable example.

6.2.2 Sampled Profiling

As discussed before we try to maximize the speed of our profiling phase by sampling aggressively. This poses another interesting trade-off: either we apply the model to every individual micro-trace and add the estimated number of cycles for each micro-trace, or we first combine the profiles of the micro-traces to a single profile, and apply the model on the combined average profile. An intermediate solution would be to group every few micro-traces in a combined profile, apply the model on each group, and then add the cycle estimates across all groups.

To find the optimal sampling settings, we perform the following exhaustive experiment. We evaluate window sizes of 1M, 10M, 100M and 1B instructions, resulting in 1000, 100, 10 and 1 window(s), respectively, for the instruction traces of one billion instructions. For each of these window sizes, we profile micro-traces of 1K, 5K and 10K instructions, and multiple sample rates, such that we profile at least 100K instructions of the 1B instructions trace. This approach of using a fixed number of profiled instructions is visualized in Figure 6.2. In the top half, one micro-trace is profiled for each window. In the bottom half, two micro-traces are profiled in each window, but the window size doubles, which leads to the same amount of profiled instructions.

Figure 6.3 shows the average error of the performance model for all of the experiments, with on the horizontal axis the total number of profiled instructions (e.g., a window size of 1M instructions, micro-traces of 1000 instructions and a sample rate of 1, results in 1M instructions profiled out of 1B as shown by the lowest blue dot). Intuitively, the more instructions are profiled, the slower the profiling step. However, the slowdown is not linearly proportional

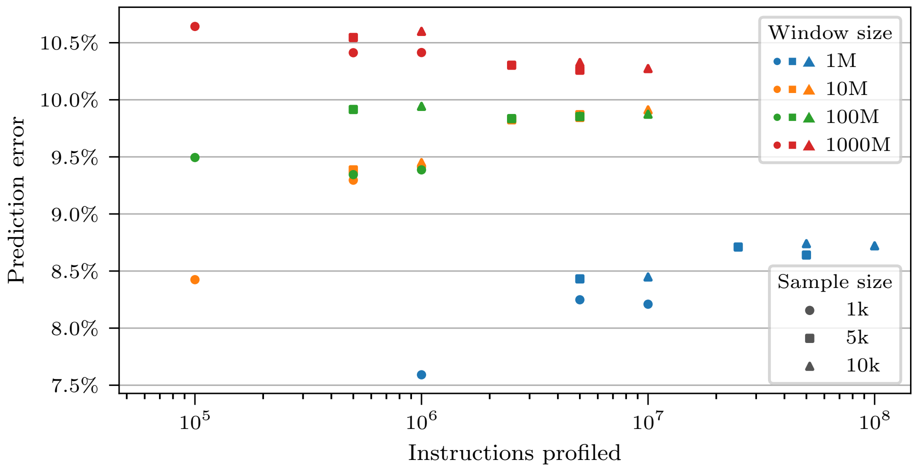


Figure 6.3: Average absolute prediction error versus number of instructions profiled. Different colors represent different window sizes; the symbols reflect the sample sizes (see legend).

to the number of instructions, because of the fast-forwarding overhead and the overhead for storing the profiles. In terms of the size of the profile, the smaller the window size, the larger the profile, because we need to keep a profile for every individual window.

The lowest error (7.6% visualized by the blue point at the bottom) is obtained for a 1M instruction window and 1K micro-trace at a sample rate of one micro-trace per window. Note that this is the sampling configuration we used to obtain all previous and next results. If we want to sacrifice some accuracy for faster profiling (8.5% error, leftmost orange point), a window size of 10M instructions with a 1K micro-trace and sample rate of one micro-trace per window may be a good alternative.

Clearly, decreasing window size improves accuracy. Being able to track short-term phase behavior is important to obtain better accuracy. This can be attributed to the fact that some of our modeling techniques, such as the contention modeling and the modeling of chained LLC hits, rely on characteristics of specific sequences of instructions, which get averaged when the window size or the sample size is too large.

Note that this also explains why extracting longer or more micro-traces from a window does not necessarily decrease the error. For example, extracting 10 micro-traces of 1K instructions or one micro-trace of 10K instructions from a window of 1M instructions causes a 0.5% *increase* in error versus extracting 1 micro-trace of 1K instructions. The explanation for this counter-intuitive behavior is that applications may suffer from high functional unit contention during small phases of just a few hundred instructions, and low contention for the rest. Since we extract many micro-traces, this extreme behavior is already present in some of the micro-traces. However, if we extract longer or more micro-traces in one window, the behavior of these few hundred instructions is

Component	Arch-dep	I-cache	Branch	LLC-chain	D-cache	MLP + queue
Avg error	6.7%	7.0%	7.0%	7.0%	7.3%	7.6%
Max error	20.7%	20.8%	21.7%	21.7%	22.0%	22.3%

Table 6.2: Average and maximum error of introducing a new micro-architecture independent component.

averaged out with the other instructions. Because of this averaging effect, we lose the ability to model fine-grained contention, leading to slightly worse performance predictions. Predicting performance for all micro-traces individually, i.e., having smaller windows, would likely improve accuracy, but it would also largely slow down the model evaluation time and increase the size of the profile.

To illustrate why combined micro-traces can lead to worse results, imagine two micro-traces of 1000 instructions. The first contains 300 loads and the second 200 loads. Following Equation 3.10, the micro-traces can execute at an effective dispatch rate of 3.33 and 4, respectively, assuming the physical dispatch width is 4. This yields execution times of 300 and 250 cycles, respectively. If we combine the micro-traces, the effective dispatch rate due to the load instructions would be $\frac{2000}{500} = 4$. Thus in the separate case, the total cycle count is 550 and in the combined case it is 500, a difference of 10%. This shows that, because contention in the functional units can happen at small time-scales, contention has to be modeled using separate micro-traces (which are not too long).

6.2.3 Micro-Architecture Independent Modeling

One of our contributions is to make the profile independent of the micro-architecture, such that we require only one profiling step to model a large range of micro-architectures. We do this by proposing models that predict the number of cache and branch misses, MLP, memory bandwidth usage, chained LLC hits, and functional unit contention, based on a micro-architecture independent profile. Because each of these models introduces additional inaccuracies, we expect the error of a micro-architecture independent model to be higher than that of a micro-architecture dependent model. In this section, we show how much error each of the components of the micro-architecture independent model introduces.

We start with a model similar to the original interval model [32], where we extract cache miss rates, branch miss rates, the amount of MLP, and memory bus queuing time from detailed Sniper simulations. The profile only contains the instruction mix and instruction dependency information (critical dependence path, average dependence path, load dependence distribution). We already incorporate the improved functional unit contention modeling and the LLC hit chain modeling. This model (denoted Arch-dep in Table 6.2) has an error of 6.7% on average across all benchmarks for our reference architecture. Next, we gradually add each of the architecture-independent components, see Table 6.2. The I-cache column shows the error when using the instruction

Parameter	Low-end	Middle	High-end
Dispatch width	**2**	**4**	**6**
ROB entries	32 - 48 - **64**	96 - **128** - 160	128 - 192 - **256**
Instruction queue	**1/3** of the ROB size		
Branch predictor	**pentium-M** [**65**] - gshare - global		
L1-I cache	**32 KB, 4-way associative, latency 1 cycle**		
L1-D cache	**32 KB, 8-way associative, latency 4 cycles**		
L1-MSHR	**10 entries**		
L2 cache	128 KB - **256 KB** - 512 KB		
	8-way associative, latency 8 cycles		
L3 cache	1 - 2 - **4** MB	4 - 6 - **8** MB	8 - 12 - **16** MB
	16-way associative, latency 30 cycles		
Memory bandwidth	**8 GB/s**		
Memory latency	**120 cycles**		

Table 6.3: Core configuration design space. Default values for the low-end, middle, and high-end cores are indicated in bold. The middle bold configuration is our reference architecture.

cache miss rate from StatStack instead of simulation, which increases the average error by 0.3%. Adding the micro-architecture independent branch predictor model does not noticeably increase the average error, which is also the case for using the LLC hit rate from StatStack to model LLC hit chaining. Estimating the D-cache misses (mainly the LLC misses) using StatStack introduces an error increase of 0.3%, and a similar increase is incurred by modeling the MLP and memory queuing in a micro-architecture independent way. Overall, micro-architecture independent modeling increases the error by 0.9% on average, while the maximum prediction error increases by only 1.6%.

6.2.4 Relative Accuracy across a Design Space

To show that the model is accurate across a large design space, we define a broad set of processor configurations over which we evaluate the model, see Table 6.3. We split this set into three categories: low-end (dispatch width of 2), middle (dispatch width of 4), and high-end (dispatch width of 6) cores. The range of the sizes of the different components are in the table, the default value for each category is shown in bold. The parameters that are not mentioned in the table are equal to those of the standard Nehalem core configuration as included in the Sniper distribution. There are 243 different configurations in this design space.

6.2.4.1 Speedup

To illustrate the speedup obtained by using an architecture-independent profile, we calculate how much time it takes to evaluate the full design space for all benchmarks using a SimPoint of one billion instructions per benchmark. Our profiler incurs a one-time cost of 3 minutes on average per benchmark, or 1.4 compute hours to profile all applications. Adding 5 seconds per configuration and benchmark to calculate the model, our framework only needs 11.4 hours.

Simulating one processor configuration using Sniper takes approximately 30 minutes per benchmark. This means that simulating the complete design space consisting out of 243 processor configurations and 29 benchmarks takes around 150 compute days. Hence, our micro-architecture independent model can evaluate the processor design space 315× faster.

Evaluating the same design space using the previously proposed interval model requires multiple functional simulations. It requires 3 branch predictor simulations, 21 cache hierarchy simulations (3 different L2 sizes combined with 7 different L3 sizes) and the associated 21 MLP simulations. Note that for the MLP-simulations, we assume that the MLP can be calculated concurrently for multiple ROBs as proposed by Eyerman et al. [30]. Based on the speed of the functional cache simulation of Sniper, we assume that the combination of all these functional simulations can run at 1.5 MIPS. Hence, the total time required to complete all functional simulations is 200 hours. Assuming the model evaluation takes 1s per processor-application combination, evaluating the design space takes 202 hours. This means that our model is 18× faster compared to the previously proposed interval model.

6.2.4.2 Relative Accuracy

While we showed that the approach of evaluating micro-traces for every window separately is sound, we further confirm this for a complete design space of processors. We compare the separate evaluation with the other extreme approach: evaluating the model for all micro-traces combined. We extract micro-traces of 1000 instructions from instruction windows containing 1M instructions and evaluate them both separately and combined. These sampling approaches were previously visualized as the blue and red dot at x-coordinate 10^6 in Figure 6.3, respectively.

Figure 6.4 visualizes the prediction errors for the complete design space and both sampling approaches using a cumulative distribution function. The average absolute error for evaluating all processor designs and benchmarks using separate micro-traces is 9.3% (compared to 13.3%). The maximum prediction error is 40.0% when evaluating the micro-traces separately, which is significantly lower than the 70.8% when evaluating combined micro-traces. This clearly indicates that the evaluation of micro-traces separately is the best approach.

Secondly, we show a box-and-whisker plot to break down the performance prediction error per benchmark in Figure 6.5. The box is the range between the first and third quartile, the horizontal line in the box is the mean, the dot is the median, the whiskers cover all points within 1.5 inter-quartile distances outside the box, and the points represent the outliers. For 23 of the 29 benchmarks the distance between the first and third quartile is 10% or less. Furthermore,

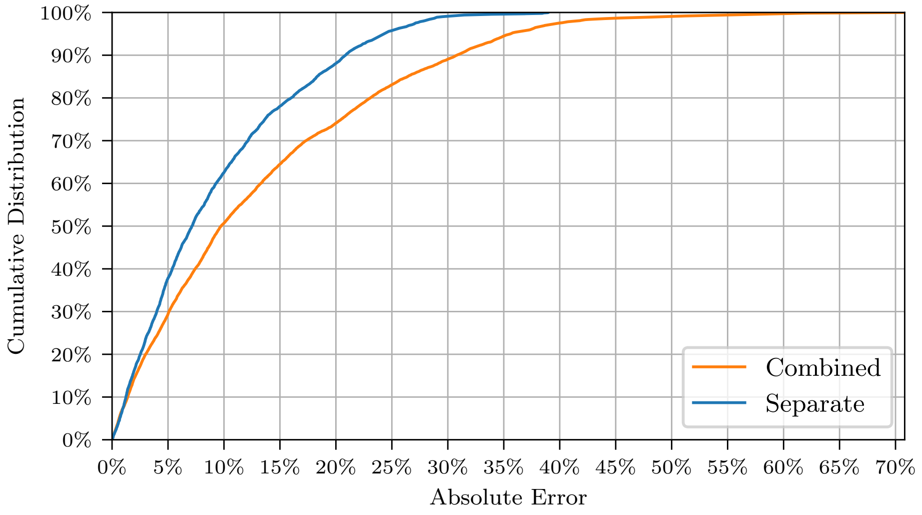


Figure 6.4: Cumulative distribution of the performance prediction error when evaluating micro-traces separately versus combined.

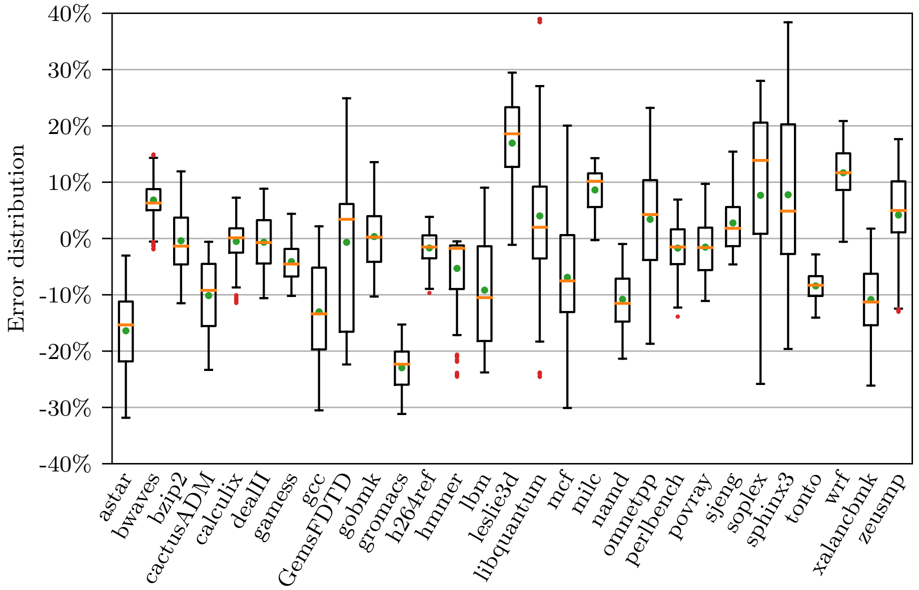


Figure 6.5: Box-and-whiskers plot showing the performance prediction errors for each of the SPEC CPU 2006 benchmark for the complete design space.

most boxes are close to 0% indicating good absolute accuracy. The biggest outliers, in casu *GemsFDTD*, *libquantum*, *mcf*, *soplex* and *sphinx3*, feature a big DRAM component relative to the base component, which is the most difficult component to predict accurately.

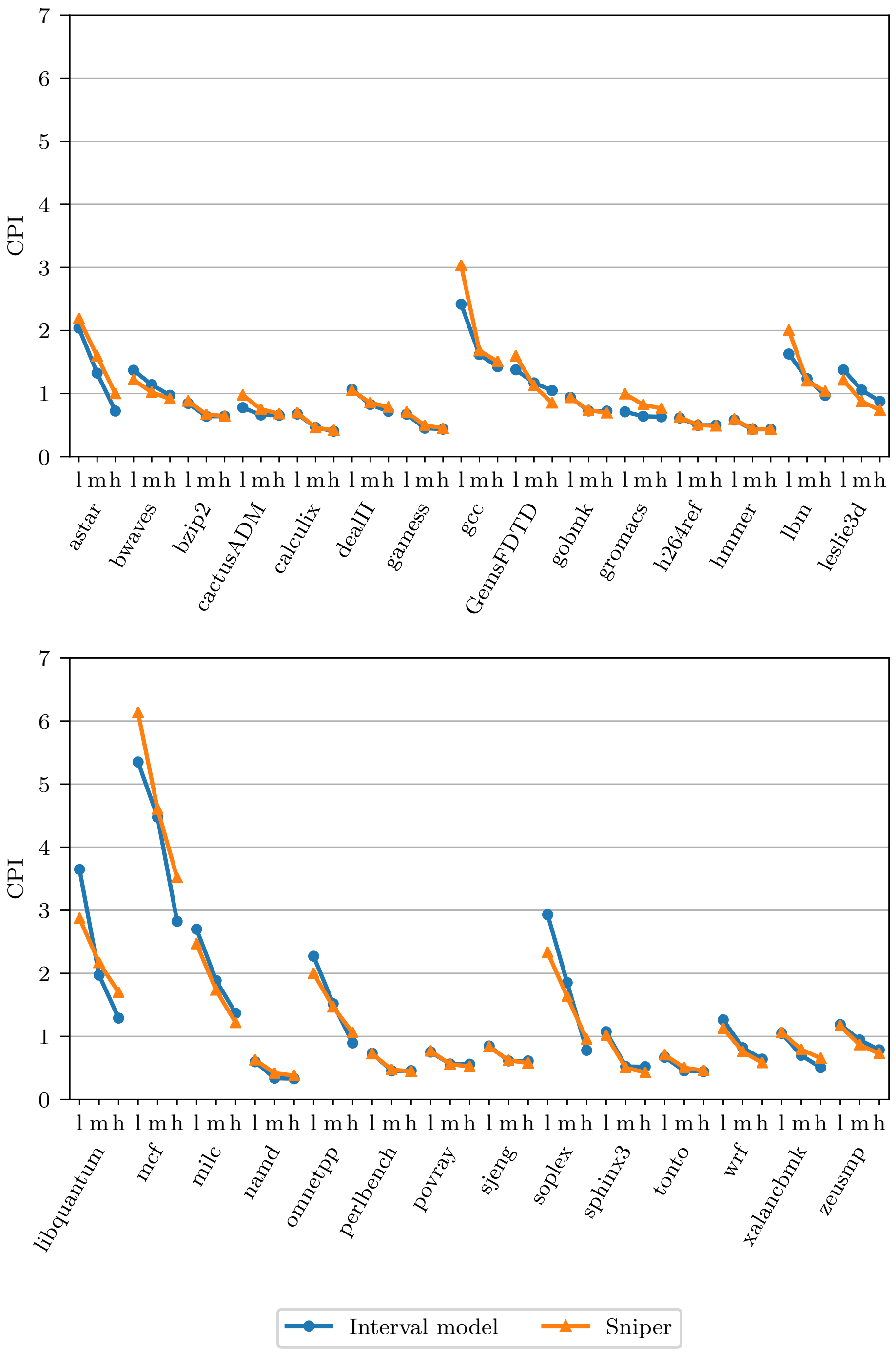


Figure 6.6: Micro-architecture independent interval model CPI estimation versus simulated CPI using the Sniper detailed simulator, for three core configurations (l=low-end, m=middle, h=high-end).

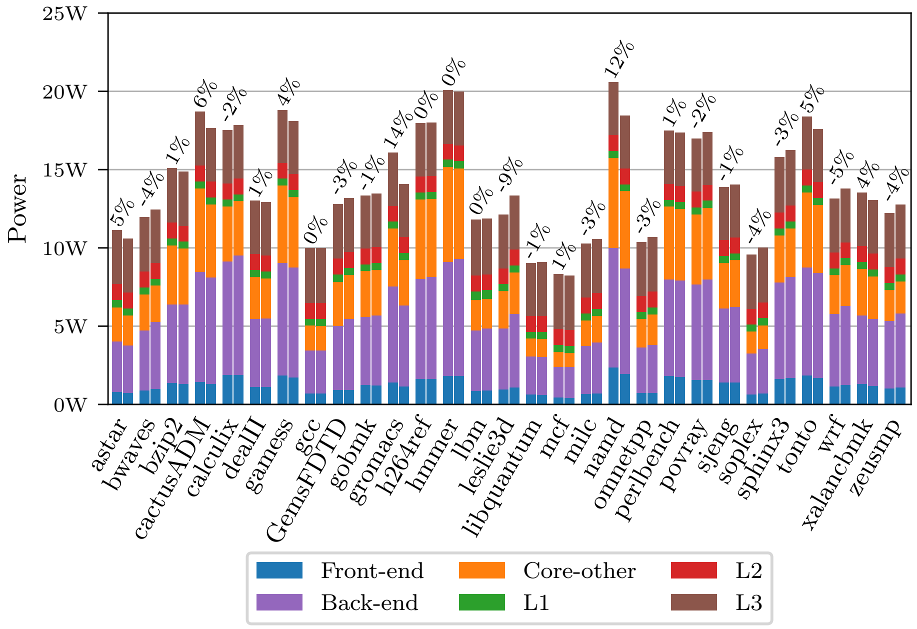


Figure 6.7: Power stacks generated by the model (left bar) and by Sniper (right bar), and the error of the model versus Sniper simulation (top).

Finally, as this is not obvious looking at the box-and-whiskers plot, we show that our model exhibits a good relative accuracy in Figure 6.6. This means that it is able to track performance trends between different processors well. Figure 6.6 shows the CPI for the three bold configurations in Table 6.3 as estimated by the micro-architecture independent interval model and as simulated by Sniper. For most benchmarks, the model estimations are very close to the simulated values. For other benchmarks (e.g., *gcc*, *lbm*, *mcf*, and *sphinx3*), the predictions errors are somewhat higher, but the model still tracks the performance trend well. This is the most important feature of our model as it enables design space explorations. We will discuss this in more detail in Section 7.4, and show more detailed results.

6.3 Power Prediction

6.3.1 Absolute Accuracy

We first evaluate the power consumption predicted by McPAT in combination with our interval model for the reference architecture described in Table 6.1. Similar to CPI stacks, we can also build power stacks showing the power consumption for different parts of the processor. The power stacks and the prediction error of the model are shown in Figure 6.7. The visualized components are the power used by the front- and back-end, other core power (core-other) and each level of the cache hierarchy. The front-end component includes, among others, the instruction-fetch logic and branch predictor, while the back-end combines the power consumed by the functional units, ROB, etc. Core-other is used for the remaining structures which McPAT does not list in detail. The stacks take into account both static power and dynamic power consumption.

As Figure 6.7 shows, the maximum prediction error is 14% for the *gromacs* benchmark. This is to be expected as this is the benchmark featuring the largest error for our performance predictions too, making it difficult to accurately predict dynamic power consumption. The average absolute error for our reference architecture across all SPEC CPU 2006 benchmarks is 3.4%.

6.3.2 Relative Accuracy across a Design Space

Similar to the performance predictions, we again show that our model can accurately predict trends, this time for power consumption. We use the same design space as defined in Table 6.3.

First, we show that using the performance predictions with separate micro-traces instead of combined micro-traces also provides good prediction accuracy for the power predictions. Note that we do not evaluate the power consumption per micro-trace, but only use the performance predictions obtained from using separate micro-traces. The reason for this is that the prediction of dynamic power only relies on activity factors. The consequence is that it does not matter whether the activity factors are calculated over separate or combined micro-traces as is the case for the performance.

To show the rationale behind this, we rely on the same example as for our performance predictions, namely two micro-traces containing 300 and 200 loads, respectively. Remember that this yielded execution times of 300 and 250 cycles, respectively. Calculating activity factors using Equation 3.16 gives activity factors of $\frac{300}{300} = 1$ and $\frac{200}{250} = 0.8$, respectively. Thus, the average activity factor, respecting the execution times, is $1 \times \frac{300}{550} + 0.8 \times \frac{250}{550} \approx 0.91$, which is exactly equal to the activity factor if we add the load-instructions from micro-traces together: $\frac{500}{550} \approx 0.91$.

Figure 6.8 shows that, as expected, power predictions across the complete design space are better when using the performance predictions from separate micro-traces. The average absolute error is only 4.3% using the performance predictions from separate micro-trace evaluation, compared to 7.1% when using performance predictions from combined micro-traces. Similar to the error distribution of the performance predictions, the outliers are largely eliminated, reducing the maximum error of 48.6% to 21.5% and thus shifting the complete distribution curve to the left.

A breakdown of the distribution of the power predictions errors per benchmark, using a box-and-whiskers plot, is shown in Figure 6.9. The box-and-whiskers plots is built following the same rules as in Figure 6.5. With respect

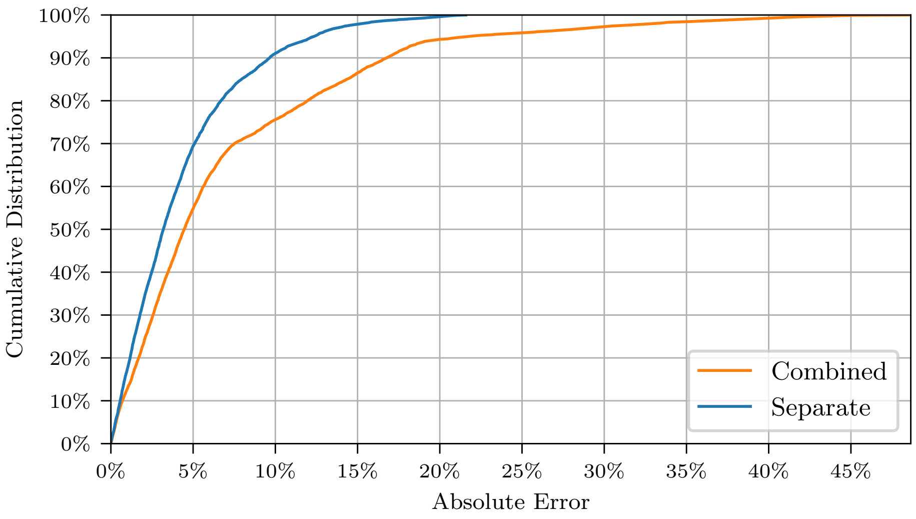


Figure 6.8: Cumulative error distribution for power predictions, comparing evaluation of the model using combined and separate micro-traces.

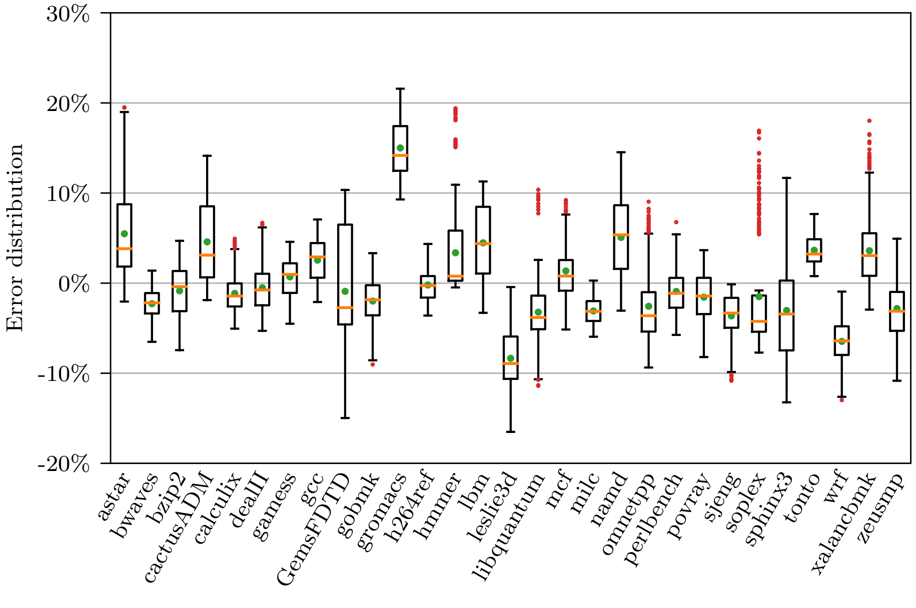


Figure 6.9: Box-and-whiskers plot showing the power prediction errors for each of the SPEC CPU 2006 benchmark for the complete design space.

to the power predictions, all benchmarks exhibit an error spread of less than 10% between the first and third quartile. Note however that, e.g., for the *soplex* benchmark, the number of outliers is significant (close to 25% of the whole design space). This can be attributed to the large spread of the performance prediction errors as was shown in Figure 6.5.

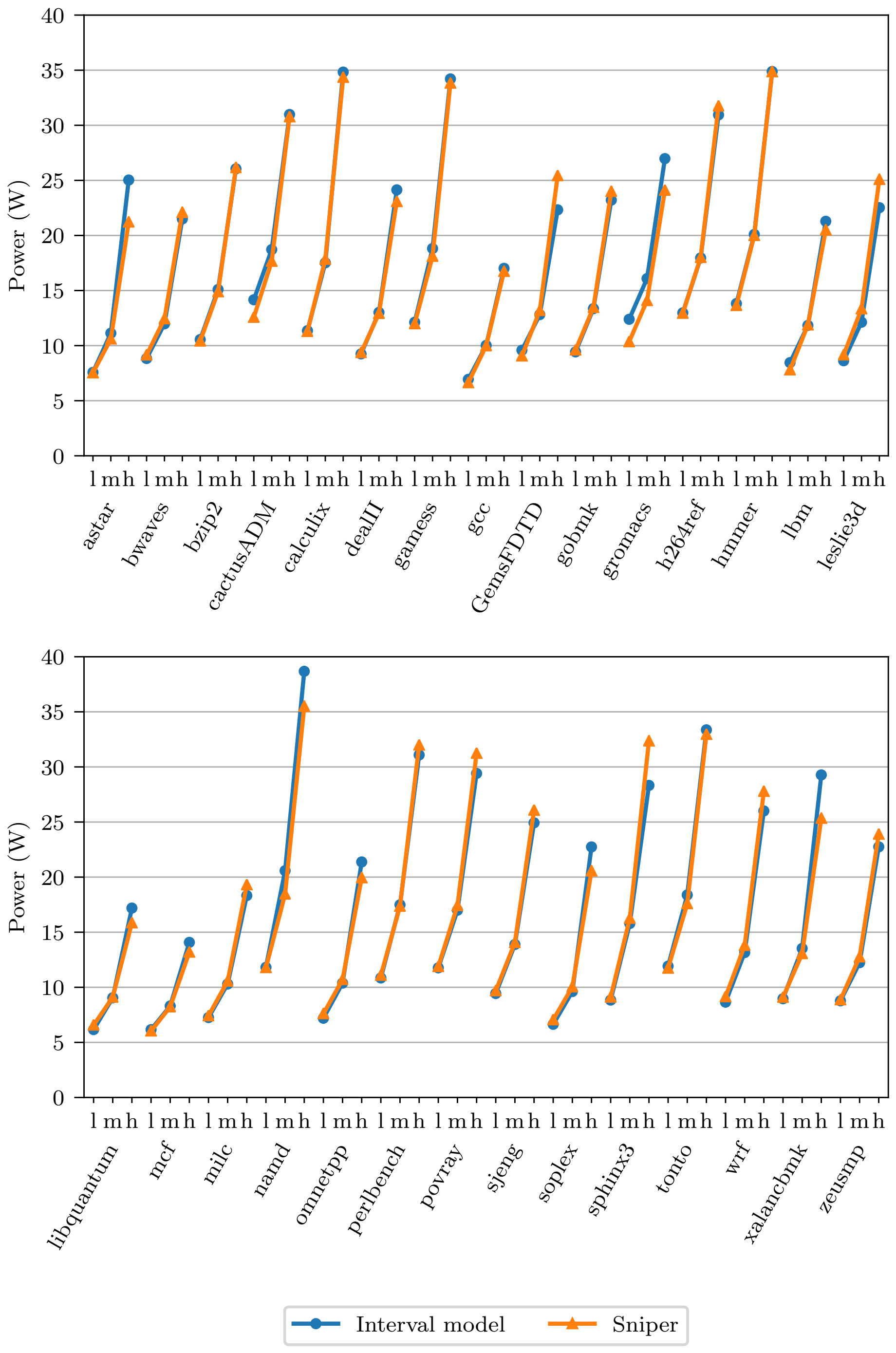


Figure 6.10: Micro-architecture independent interval model power estimation versus simulated power using the Sniper detailed simulator, for three core configurations (l=low-end, m=middle, h=high-end).

Furthermore, Figure 6.10 shows the power estimations for the low-end, middle and high-end configurations shown in Table 6.3 in bold, compared to Sniper-simulations using McPAT to predict power consumption. Although the three configurations are fixed, power consumption varies significantly both over the different benchmarks, due to different activity factors, and over the different processor configurations. For example, the power of the high-end core ranges between less than 15 W to above 35 W. The power difference between a low-end and high-end core executing the same benchmark is often bigger than 15 W. Note that, for most benchmarks, the absolute power predictions by the model are accurate, and even if there is a prediction error, such as in the case of *namd*, the model tracks the trend between different processor architectures accurately.

6.4 CPI Stacks

One of the most useful features of the interval model is that it enables building detailed CPI stacks. Since the model is a summation of different penalty terms, we can visualize each of them. We show four interesting CPI stacks in Figures 6.11 and 6.12 to illustrate the conclusions one can draw from them. CPI stacks for all SPEC CPU 2006 benchmarks can be found in Appendix A.

To gain more insight, we split the base component into four different sub-components. These sub-components follow the minimizing operation from Equation 3.10 to calculate D_{eff}. Hence, if the physical dispatch width is the minimum and thus the most limiting factor, only the base component is visible. If there are stricter limits due to dependences, issue ports or functional units, this is shown as an extra stack on top. To calculate this extra stack, we divide the number of micro-operations by the respective dispatch rate and subtract the preceding components as shown in the below equations:

$$Base = \frac{N}{D_{physical}} \tag{6.1}$$

$$Crit = \max\left(0, \frac{N}{D_{critical}} - Base\right) \tag{6.2}$$

$$Port = \max\left(0, \frac{N}{D_{port}} - Critical - Base\right) \tag{6.3}$$

$$Unit = \max\left(0, \frac{N}{D_{unit}} - Port - Critical - Base\right) \tag{6.4}$$

The eight different components (for which the first four components are the base sub-components) visualized in the CPI stacks are described below:

- Base component (calculated following Equation 6.1): The minimum execution time equals the number of micro-operations divided by the physical dispatch width.

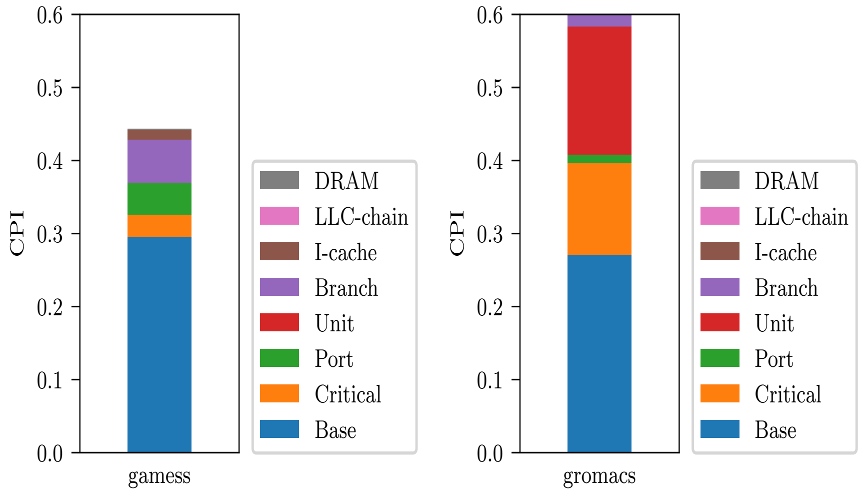


Figure 6.11: Comparison of the base component for the *gamess* and *gromacs* benchmarks.

- Critical component (calculated following Equation 6.2): The penalty introduced due to long dependence chains which lower the processor's issue and execution rate.
- Port component (calculated following Equation 6.3): The penalty introduced because multiple functional units are connected to the same issue port, thus requiring multiple cycles to issue all micro-operations.
- Unit component (calculated following Equation 6.4): An extra penalty for instructions that have to be executed by non-pipelined functional units.
- Branch component (second term in Equation 3.1): The penalty for the number of branch mispredictions multiplied by the sum of the front-end refill time and the branch resolution time.
- I-cache component (third term in Equation 3.1): The penalty due to the number of I-cache misses times the latency.
- DRAM component (fourth term in Equation 3.1): The penalty for the number of LLC misses times their respective latency and divided by the MLP.
- LLC-chain component (fifth term in Equation 3.1): The penalty component indicating that the ROB blocks due to multiple LLC hits on the same dependence path.

The first two selected CPI stacks, in Figure 6.11, are from the *gamess* and *gromacs* benchmark to highlight the insight one can get by splitting up the base component. The execution for the *gamess* benchmark is close to perfect because there are only a few small penalty components due to some contention

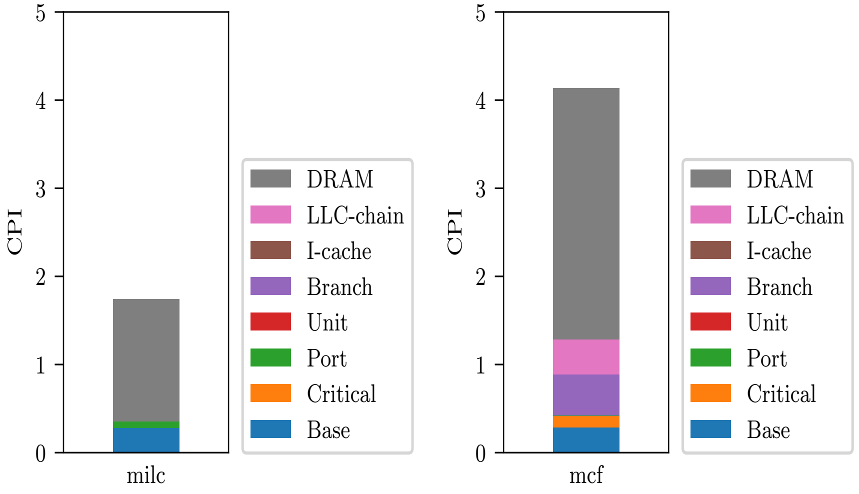


Figure 6.12: Comparison of the DRAM component for the *milc* and *mcf* benchmarks.

in the issue stage and branch mispredictions. Its performance is only 20% worse compared to the perfect execution time. For the *gromacs* benchmark, on the other hand, there is a clear problem with the issue stage. It has both a relatively large critical and unit component. The unit component is interesting because this indicates that there are instructions that have to be executed by non-pipelined functional units. In the case of our reference architecture only the functional unit that processes division and square root operations is not pipelined. This is especially interesting because the instruction mix contains merely 3% division and square root operations. Despite this small share, the influence on total performance is significant because of the long latency compared to other instructions (typically around 20 cycles per instruction). Note that in reality, the problem with issue contention is even worse as we underestimate the execution time of this benchmark by 20%.

The latter two CPI stacks, in Figure 6.12, are for the *milc* and *mcf* benchmark. Both have a large DRAM component which indicates a lot of LLC cache misses. They have an MPKI of 22 and 46, respectively. However, despite the number of misses for *mcf* only being double the number of misses for *milc*, it is $\sim 2.8\times$ slower than *milc* instead of only $\sim 2\times$. This shows that next to a lot of misses, *mcf* also exhibits poor MLP. The (well-known) reason for this is that *mcf* is a pointer-chasing benchmark with a lot of load misses depending on each other.

Another interesting feature of CPI stacks is that it allows one to compare different architectures. As an example we show the *gromacs* benchmark again for our reference architecture and the low-end, low-power architecture shown in bold in Table 6.3. Figure 6.13 shows that despite doubling the dispatch width and cache size, the performance gains are minimal at only 13%. This is of course due to the fact that even though the processor can dispatch twice

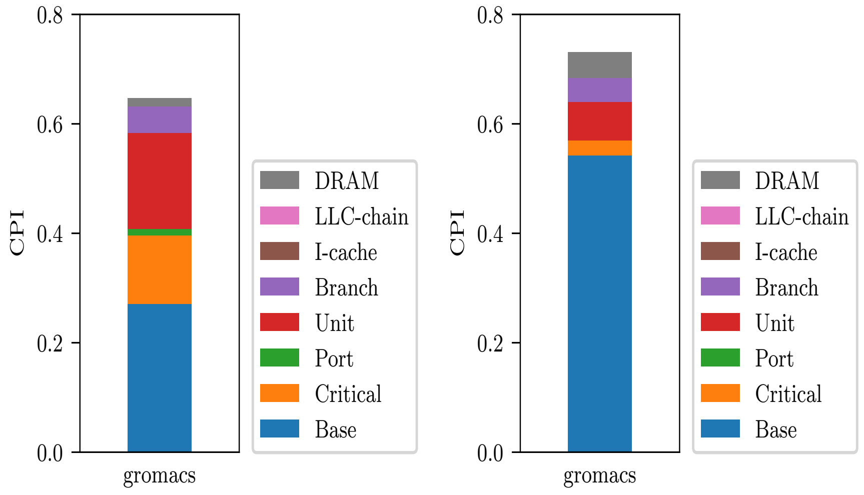


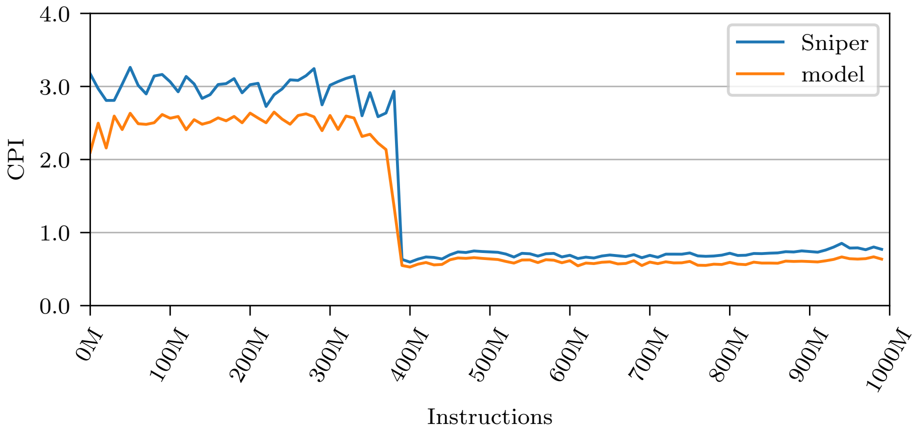
Figure 6.13: Comparison of the performance of the *gromacs* benchmark on our reference architecture (left) and a low-power architecture (right).

the number of instructions in one cycle, the issue stage cannot process all of the division operations fast enough to keep up with the doubled dispatch rate. Meanwhile, the power consumption of the medium processor compared to the low-end is about 1.5× for the *gromacs* benchmark. Hence it is questionable whether the performance gain is worth the increase in power consumption. Using our model, this decision can be made multiple times faster than using detailed simulation.

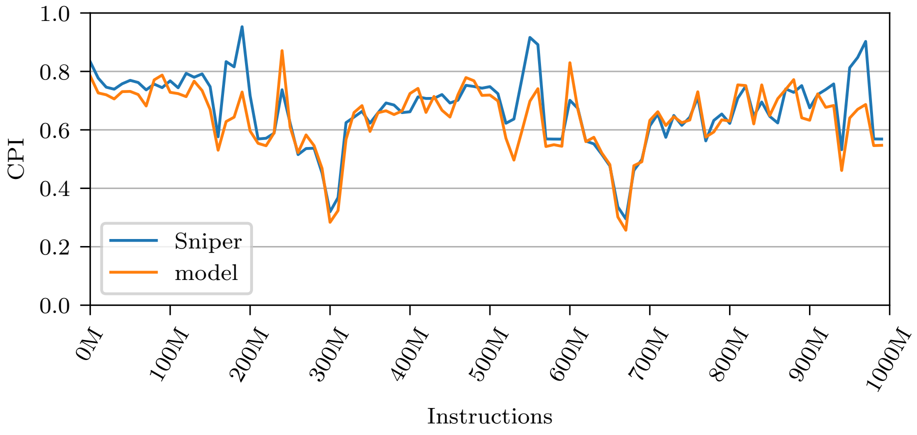
6.5 Phase Analysis

Another advantage of evaluating the model on a per-micro-trace basis instead of on combined micro-traces is that it allows us to study phase behavior. This is a useful feature because it is possible that an application exhibits very different behavior throughout its execution. For example, an application may first load its data in memory leading to a memory-intensive phase first and afterwards perform lots of computation leading to a compute-intensive phase. By offering insight into this phase behavior, it may be possible to exploit application characteristics through phase-aware optimizations in hardware or software.

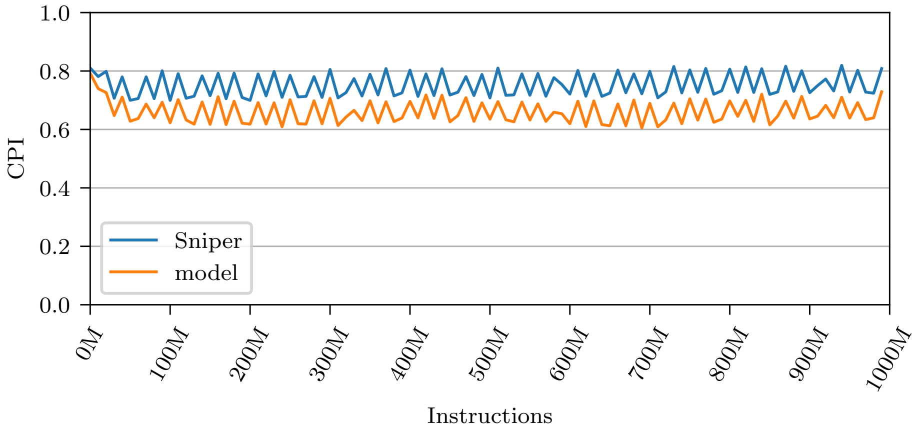
Figures 6.14a through 6.14c show the CPI variation for three benchmarks per 10 million instructions, for both Sniper and the model. Phase graphs for all benchmarks are provided in Appendix B. We show the average error across the whole execution, and the *phase accuracy coefficient (PAC)* to quantify how well we predict execution time and track phase behavior. We define PAC as the average absolute error of the relative difference between two consecutive



(a) *astar* – error: -16.9% – PAC: 6.2%



(b) *bzip2* – error: -4.3% – PAC: 6.7%



(c) *cactusADM* – error: -12.2% – PAC: 1.9%

Figure 6.14: Phase graphs for the *astar*, *bzip2* and *cactusADM* benchmarks from the SPEC2006 CPU benchmark suite.

intervals of one million instructions:

$$PAC = \frac{1}{N} \sum_{i=1}^{N} \left| \frac{C_S(i-1) - C_S(i)}{\overline{C_S}} - \frac{C_M(i-1) - C_M(i)}{\overline{C_M}} \right| \tag{6.5}$$

$C_S(i)$ is the number of cycles predicted by Sniper for interval i, whereas $C_M(i)$ is the number of cycles predicted by the model. $\overline{C_S}$ and $\overline{C_M}$ are the average number of cycles of one interval over the full trace, for Sniper and the model, respectively. The PAC-values are included in Figures 6.14a through 6.14c, along with the error on the predictions of the overall execution time. An example of an application where there is a clear memory and compute intensive phase is the *astar* benchmark, see Figure 6.14a. The first 400 million instructions execute at a CPI of around 3, while the remainder of the application exhibits a CPI of 0.5, indicating that the application first mainly loads data and then performs computations on it. The model tracks an application's time-varying execution behavior well for most of the benchmarks, see for example *bzip2* in Figure 6.14b. For some benchmarks, e.g., *cactusADM* in Figure 6.14c, we observe that the PAC is lower than the overall execution time prediction error; in spite of the relative modeling error offset, the model tracks the application's phase behavior fairly well.

6.6 Stride MLP and Prefetch Results

When considering a full application, there will be few cold misses which renders the cold-miss MLP model less useful. After all, this MLP model relies on modeling burstiness through cold misses and conflict/capacity misses are assumed to be uniformly distributed. Since this is not necessarily true, we proposed the stride MLP technique as discussed in Section 4.5. In this section we evaluate the accuracy difference between the two MLP techniques. Furthermore, because all contemporary processors feature one or more prefetchers, we also evaluate our model when there is a simple stride prefetcher present.

In all of our previous simulations the prefetcher was disabled in the Sniper simulator. Since we did not have the time to redo all simulations, we select a design space where we only vary the ROB size and cache size. We build the design space like this because these are the only parameters that influence cache misses and thus MLP and prefetchability. The design space contains 35 processor designs, 7 ROB sizes combined with 5 different sizes for the last-level cache, as summarized in Table 6.4. Our stride prefetcher has 16 streams, meaning that it can track the last 16 different loads, and will not prefetch over virtual memory page boundaries. Our reference architecture is still the same as in the previous design space and has an ROB size of 128 entries combined with an LLC of 8 MB.

We evaluate the accuracy of both MLP models by quantifying the total time spent waiting for DRAM. In Sniper, the DRAM cycle component is measured as the number of cycles between a load miss blocking the head of the ROB

Core frequency	2.66 GHz
Dispatch width	4
ROB	64, 96, **128**, 160, 192, 224, 256 entries
L1I and L1D	32 KB, latency = 1 and 4 cycles, respectively
L2	256 KB, latency = 8 cycles
LLC	1, 2, 4, **8**, 16 MB, latency = 30 cycles
MSHR	Between L1D and L2, entries = 10
Prefetcher	stride prefetcher, streams = 16
Memory bus	Bandwidth = 7.6 GB/s
DRAM	latency = 45 ns

Table 6.4: Reference architecture, based on Intel Nehalem.

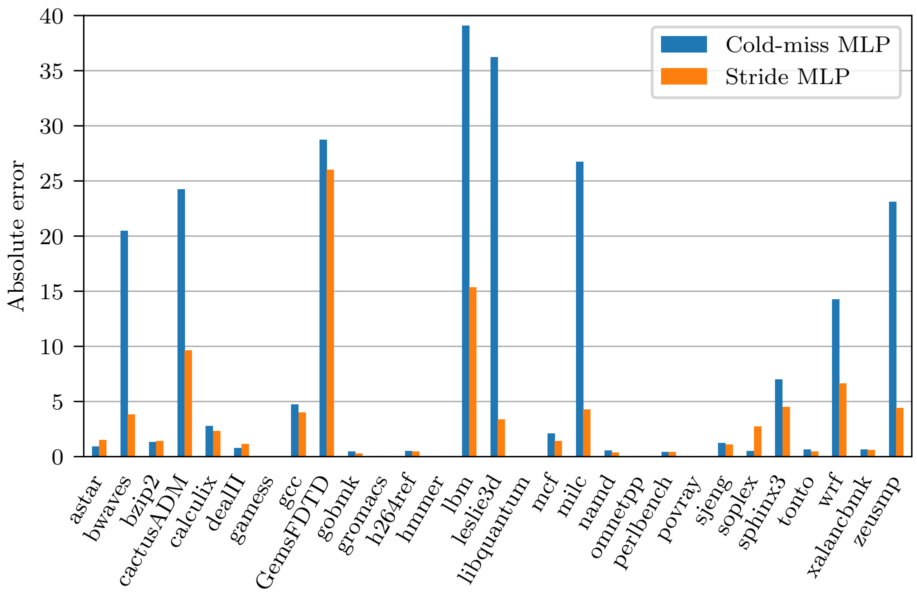


Figure 6.15: Absolute error for predicting total time waiting for DRAM for the cold-miss and stride MLP models, assuming no hardware prefetching.

and the data returning from main memory [31]. In our model, we estimate the DRAM component by multiplying the estimated number of LLC misses times DRAM access latency divided by the predicted MLP. Figure 6.15 reports the model's accuracy for predicting the DRAM waiting time against simulation. The average absolute error equals 3.3%. The highest error for the stride MLP model is observed for *gemsFDTD* (26.0%).

When running complete applications, the stride MLP model is substantially more accurate than the cold-miss MLP model which achieves an average absolute error of 8.2% and a maximum error of 39.1%. Modeling the relative

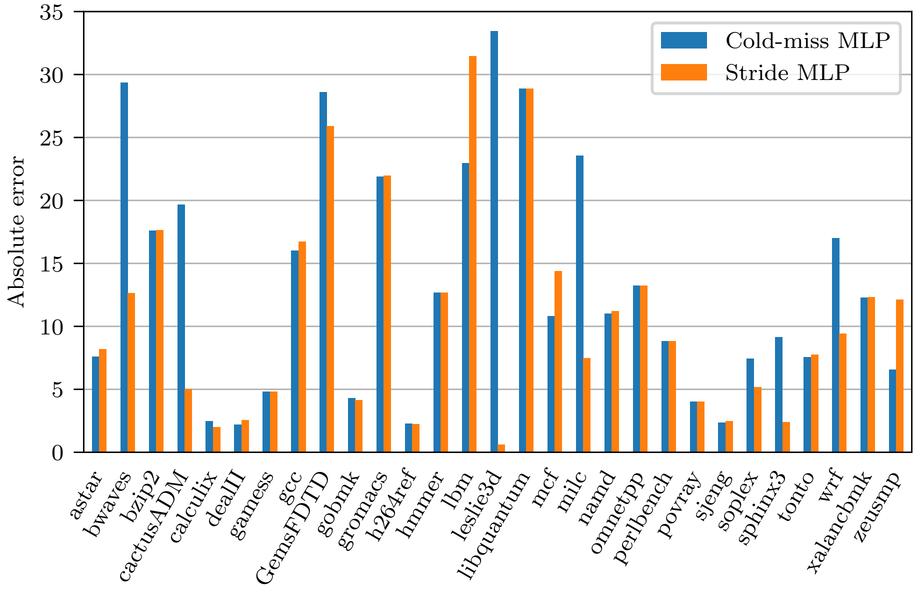


Figure 6.16: Absolute error for predicting the performance of our reference architecture for the SPEC 2006 benchmarks using the stride and cold-miss MLP models.

spacing of memory references, their dependences and strides clearly leads to a more accurate model.

If we plug the stride MLP model into the complete performance prediction model for our reference architecture, we see an average improvement of 2.8%. Figure 6.16 visualizes the performance prediction error per benchmark. The average performance prediction error are 10.6% and 13.4% when employing the stride MLP and cold-miss MLP models, respectively.

Evaluating the complete design space from Table 6.4, the average error is only 5.1% for the stride MLP model, while the average error for the cold-miss MLP model is 8.9%. Furthermore, more than 90% of the designs have an absolute error below 15% for the stride-MLP model, whereas for the cold-miss MLP model less than 80% of the designs have an absolute error below 15%. This is visualized in Figure 6.17. The largest errors are typically observed for unbalanced processor designs (e.g., a big ROB with 256 entries along with a relatively small 1 MB LLC).

Figure 6.18 reports the absolute prediction error assuming a stride-based prefetcher. The stride MLP model achieves an average absolute prediction error of 3.6% and at most 22.8% for our reference architecture. The cold-miss MLP model, which does not model stride-based prefetching, leads to an absolute average prediction of 16.9% and absolute errors up to 118%. This re-emphasizes the importance of incorporating the impact of hardware prefetching in an analytical MLP model.

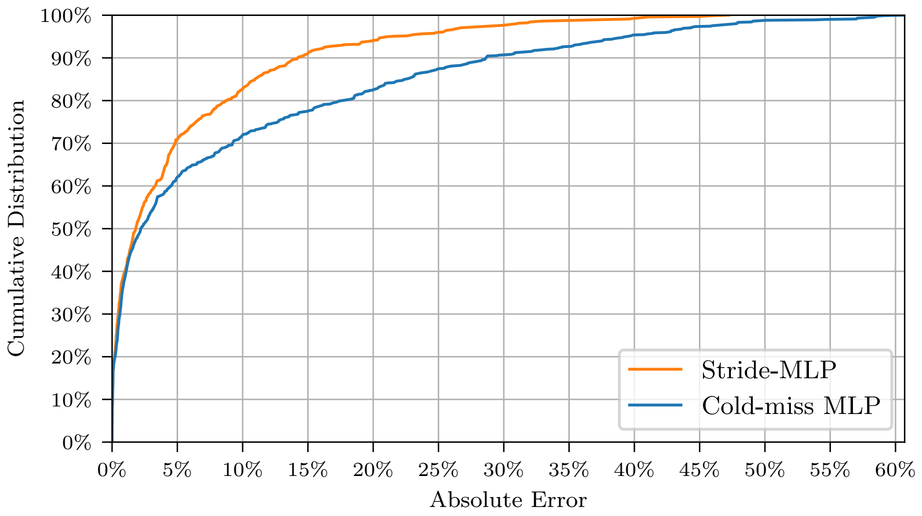


Figure 6.17: Cumulative error distribution for the DRAM prediction error when using the stride MLP model versus the cold-miss MLP model.

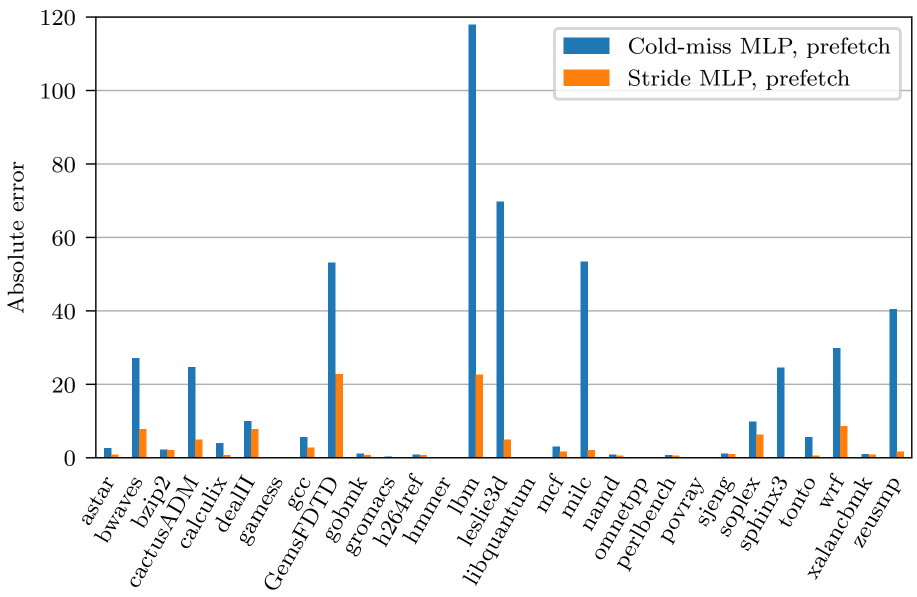


Figure 6.18: Absolute error for predicting total time waiting for DRAM for the stride and cold-miss MLP models, assuming hardware stride prefetching.

Chapter 7

Applications

In this chapter we highlight some of the applications for which our model can be used. The most obvious use case of the interval model is to guide core optimization, both with respect to performance and power/energy efficiency. First, we show how our model can be used to fine-tune a processor in order to achieve better performance. Second, we show a use case where we determine the best processor under a certain power constraint. We then show that our model can also offer insight into DVFS optimization. We conclude by introducing Pareto plots, which allow pruning a complete design space and compare our model with a regression-based model for building these Pareto plots.

All of these applications show why the absolute error of the model is less important than the relative accuracy. The main goal of our work is to speed up comparison of multiple processor architectures and offering insight in an application's performance. Hence, it is more important one can draw objective conclusions about why one processor is better than another, and less to perfectly predict absolute performance or power numbers. Another key take-away for all of these use cases is that, while these kinds of optimizations can also be performed using simulation, using our mechanistic model is around two orders of magnitude faster for a design space consisting out of a couple of hundred of processor designs.

7.1 Understanding Processor Performance

For our first use case, we want to gain insight in processor performance and optimize it if possible. We consider the case of *libquantum*, one of the SPEC CPU 2006 benchmarks. We show CPI stacks for different processor architectures visualizing the cause of performance loss. Note that, compared to the CPI stacks in Section 6.4, we combined some components together as to make it easier to interpret the final conclusion. The left CPI stack in Figure 7.1 is measured using a Sniper simulation on a high-end configuration with 128 ROB

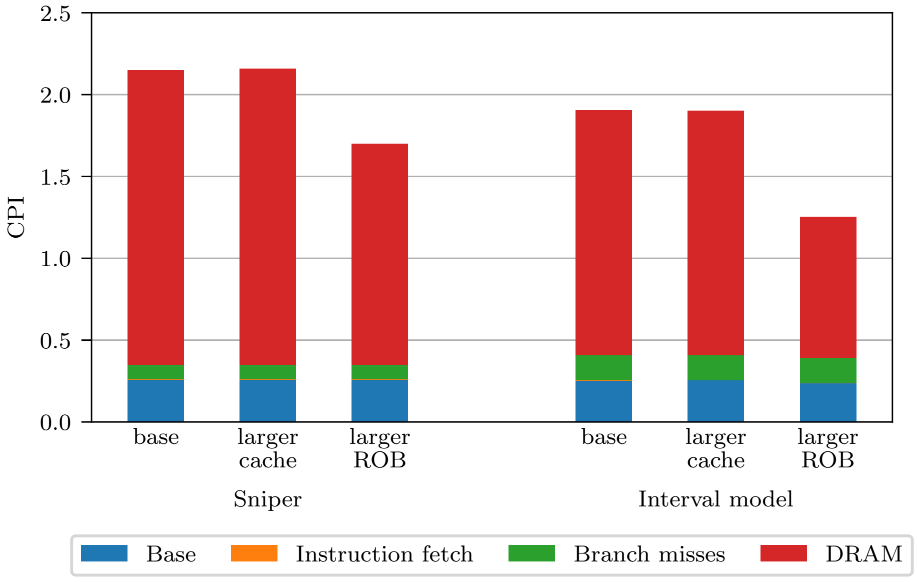


Figure 7.1: CPI stacks measured by Sniper and estimated by our model for *libquantum*. 'Base' is a configuration with 128 ROB entries and a 8 MB L3 cache. 'Larger cache' is the same configuration, but with 16 MB L3 cache, and 'larger ROB' has an 8 MB cache, but a ROB of 256 entries.

entries and a 8 MB L3 cache (base). Because there is a large DRAM component, it is intuitive to increase the cache size for better performance. However, our model (right part of the graph) shows that there is little to gain by increasing the cache to 16 MB, because the number of misses does not significantly decrease. On the other hand, increasing the ROB size to 256 entries leads to a higher MLP, effectively decreasing the DRAM component and improving performance. Even though the absolute prediction error is significant, the same trend is confirmed by simulating the larger cache and the larger ROB using Sniper. However, the key take-away here is that to reach this conclusion using simulation only, would take many hours. To arrive at the best performing processor configuration at least 3 simulations are needed, and probably more as it would not be immediately clear that increasing the ROB size leads to the largest performance improvement (e.g., an architect might first try different cache sizes, or increasing the memory bandwidth). Using our model, we can obtain the same conclusion in one profiling step and a couple of evaluations using the analytical model, which would take less than 10 minutes. To be sure, the predictions made by the model can be confirmed by detailed simulation, but this only requires a few simulations compared to exploring all possibilities using simulation.

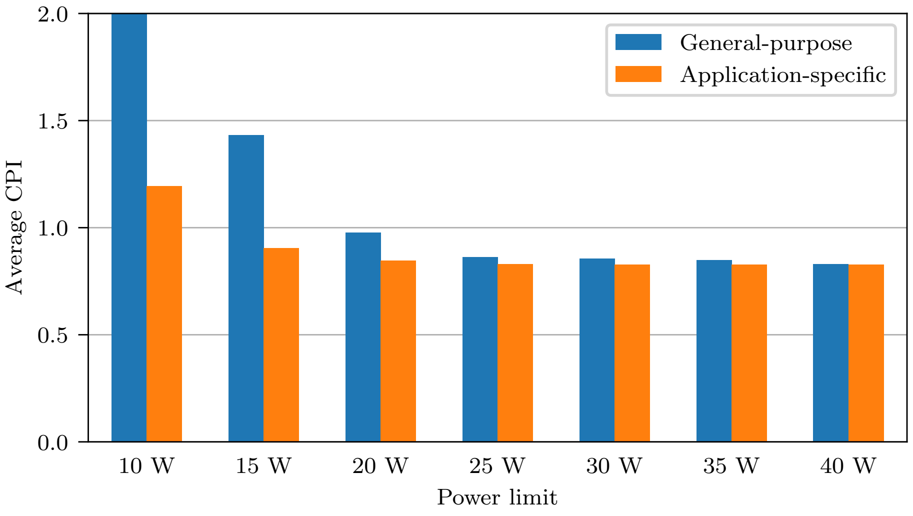


Figure 7.2: Average CPI (lower is better) when selecting a general-purpose core versus selecting an application specific core per application for different power budgets.

7.2 Optimizing Performance under Power Constraints

Processors meant to be embedded in mobile devices are usually power limited. This means that rather than delivering the best possible performance, they have to deliver the optimal performance while staying under given a power constraint. Our model can be used to efficiently explore design spaces and determine the most optimal processor configuration within given power constraints. To prove this, we set up an example design use case: we try to find the best performing configuration within various power budgets.

We first show that optimizing core configuration for individual applications indeed leads to better overall performance compared to selecting a single design that performs best on average over all applications within the same power budget. Figure 7.2 shows the average CPI (lower is better) across all applications for a single, general-purpose, optimal design (left bar) and for the application-specific designs (right bar). It is clear that selecting application-specific cores leads to higher performance (lower CPI) than selecting a single general-purpose design, especially for the low power budgets. For example, for the 15 W power budget, the single best design is a low-end design, to avoid a power overshoot for one particular benchmark, while the 28 other benchmarks can benefit from the higher performance of a middle and even high-end core, and still remain within 15 W.

The previous results were generated using the simulated results to show the potential of application-specific core design. Now, we use the model to find the optimal designs without simulating the full design space. Because there is some

Power budget	Optimal	< 1%	< 5%	< 10%	> 10%
10 W	9 (9)	6 (15)	5 (20)	3 (23)	11%,13%
15 W	6 (6)	6 (12)	12 (24)	1 (25)	11%,12%,18%,36%
20 W	10 (10)	7 (17)	10 (27)	1 (28)	12%
25 W	10 (10)	7 (17)	10 (27)	2 (29)	–
30 W	11 (11)	7 (18)	9 (27)	2 (29)	–

Table 7.1: Number of benchmarks for which the design space exploration technique results in a the optimal design, or a suboptimal design within 1%, 5%, 10% or more than 10%.

error for the power estimations, we use the model to select five possibly optimal designs out of the design space, instead of a single configuration. We do this by also selecting the best performing designs for power budgets that are 20% and 10% under the targeted power budget, as well as power budgets that are 10% and 20% higher, next to the one with the exact power budget. This method ensures that if the power consumption is somewhat over- or underestimated by the model, we can still select configurations that are under, but close to the power budget. We then simulate these five designs using cycle-level simulation, and select the one that meets the power constraints and yields the best performance. Although this technique still requires some detailed simulations, the number of simulations is drastically lower compared to exhaustively simulating the full design space.

Using this technique, we were able to find configurations that are all within the imposed power budget. Table 7.1 shows for each power budget the number of benchmarks for which we find the exact same configuration as simulating the full design space in detail. Then we show the number of benchmarks where the resulting configuration performs less than 1% worse than the actual optimal configuration, followed by less than 5% and less than 10%, and in the last column, we show how much the remaining benchmarks are off (i.e., the ones which perform more than 10% worse). Between brackets, we show the cumulative number of processor configurations that meet the performance bound. We only show results for a power budget up to 30 W, because Figure 7.2 showed that there is little to gain when using ASIPs with larger power budgets. Note that for 4 benchmarks, there is no configuration that consumes less than 10 W, explaining the smaller number of benchmarks. This was also correctly predicted by our model.

For the vast majority of the benchmarks, we find a configuration that is within 5% of the optimal configuration. The explanation for the outliers at the small power budgets is that there are a lot of configurations that are just under and above this budget. Therefore, by only picking 5 points, we sometimes miss the configurations that are just within the power budget and have the highest performance. We checked that picking more points for the 10 W and 15 W power budget leads to finding better configurations.

Frequency	1.33 GHz	2.00 GHz	2.66 GHz	3.33 GHz	3.99 GHz
Voltage	0.85 V	1.02 V	1.20 V	1.38 V	1.55 V

Table 7.2: Nehalem-based architecture using different DVFS settings.

7.3 DVFS Exploration

Dynamic Voltage and Frequency Scaling (DVFS) is commonly used to change the performance versus power consumption characteristics of a processor. By reducing frequency and voltage, a processor consumes less power, but it also yields lower performance. The impact of DVFS on performance and power consumption is application-dependent. Compute-intensive applications usually benefit more from scaling the frequency up in terms of performance, but they also tend to increase the power consumption more, because they use the core resources more intensively. Memory-bound applications see smaller performance gains with higher frequency, but usually also consume less additional power. Finding the best DVFS setting is thus application-dependent, and needs to be redone for every application and potentially for every phase within an application execution.

Our analytical model can also be used to model the impact of DVFS on performance and power consumption. In our setup, we assume that the core and the L1/L2 caches are in the same clock domain, so changing the frequency has no impact on the latency and access time in number of cycles. The LLC and main memory are in different voltage/frequency domains, so their latency remains constant in absolute time, meaning that their access time in the number of core cycles changes. The main memory access time in cycles is a direct term in Equation 3.1, and the LLC access time is part of the LLC chaining component, so we can model a frequency change. The estimated cycle count is then converted to time by dividing by clock frequency. The impact of DVFS on power consumption is modeled through McPAT, using the estimated performance.

Figure 7.3 shows the real and estimated Energy-Delay-Square product (ED^{2}P) for 5 different frequency and voltage settings (see Table 7.2) across all benchmarks. The numbers are normalized to the 2.66 GHz setting, which was the base frequency for all previous results. We choose ED^{2}P here as a metric, because the Energy-Delay product (EDP) usually prefers the lowest frequency setting thus yielding an uninteresting figure, while for ED^{2}P, the optimum varies across the benchmarks. Although there is some error on the ED^{2}P estimations (the points do not coincide), we find that our model predicts the optimal frequency for all benchmarks, except for *lbm*, *sphinx3* and *zeusmp*. The ED^{2}P-trend is predicted incorrectly for the *sphinx3* benchmark due to inaccurate modeling of the extra delay caused by accessing the LLC. For the *lbm* and *zeusmp* benchmarks, the incorrect prediction can be attributed to errors enforcing each other because the performance is underestimated while the power consumption is overestimated or vice versa.

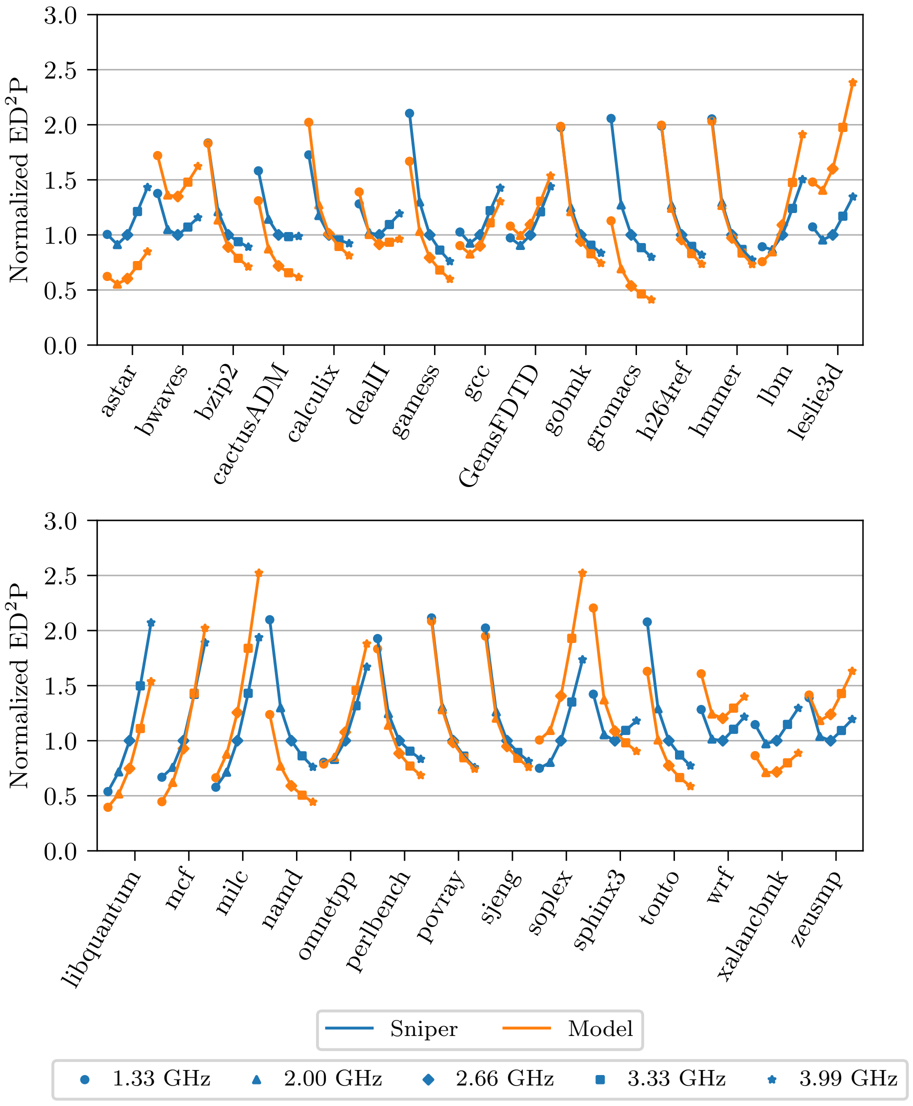
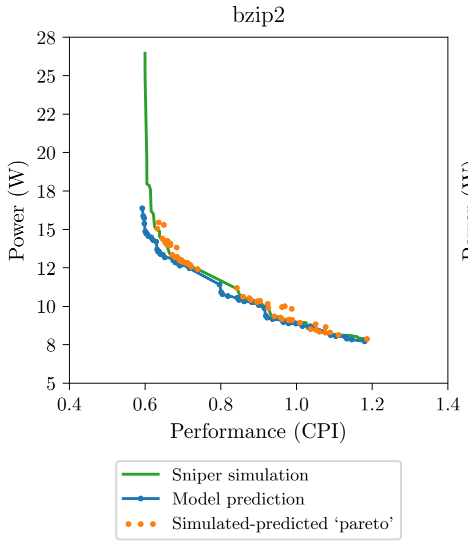


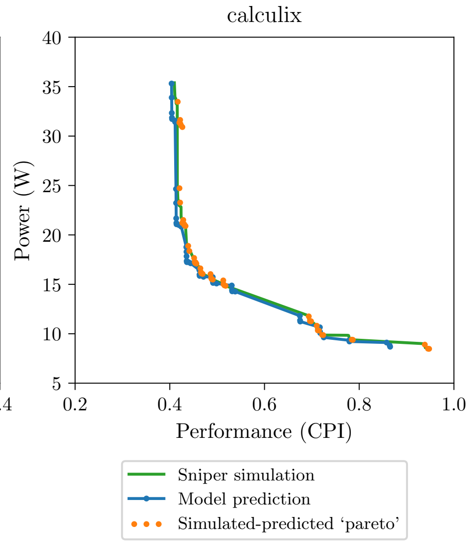
Figure 7.3: ED^{2}P for different frequencies calculated with Sniper and our model, normalized per benchmark against the Sniper configuration running at 2.66 GHz.

7.4 Pareto Curves

We can expand this performance and power design space exploration technique even further by building Pareto curves. These curves consist of the Pareto-optimal set of configurations, i.e., the set of configurations that have either higher performance or lower power consumption than any other configuration. Or in other words, there exists no other configuration that beats the Pareto-optimal configurations in both performance and power. This sec-



Error performance:	4.6%
Error power:	2.4%
Sensitivity:	42.6%
Specificity:	86.3%
Accuracy:	75.3%
HVR:	95.3%



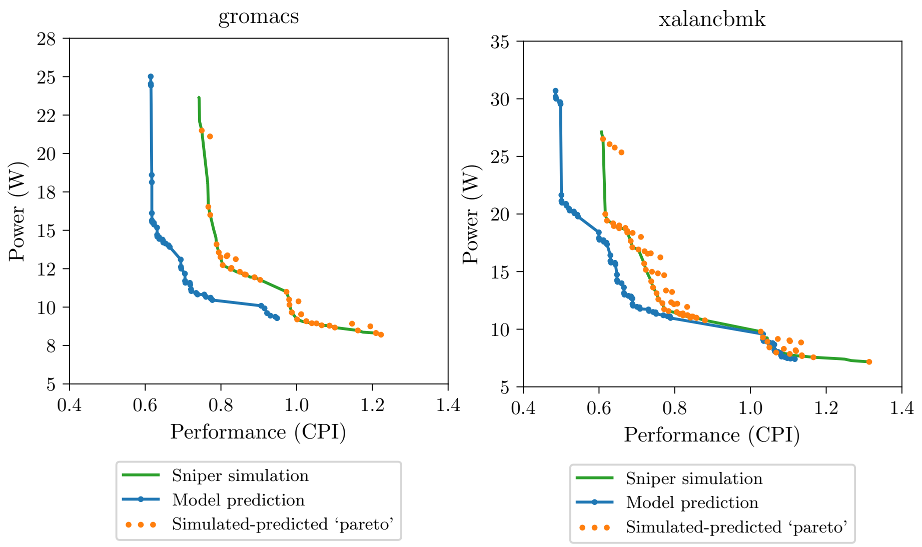
Error performance:	3.0%
Error power:	2.1%
Sensitivity:	31.9%
Specificity:	88.8%
Accuracy:	67.5%
HVR:	99.0%

Figure 7.4: Pareto frontiers for the *bzip2* and *calculix* benchmarks.

tion quantifies how well the model is able to construct a Pareto-optimal set of configurations.

Figures 7.4 and 7.5 show the Pareto frontier obtained using Sniper simulations (green) and the Pareto frontier obtained by the model (blue). The orange points are the predicted Pareto-optimal configurations (so the ones constructing the blue curve), but plotted with simulated performance and power consumption. Pareto frontiers for all SPEC CPU 2006 benchmarks are given in Appendix C. The difference between the blue and green curves shows the error of the model, while the difference between the orange points and the green curve indicates how well we can predict actual Pareto-optimal configurations.

For some benchmarks (e.g., *bzip2* and *calculix*), the blue and green curves are close, indicating high accuracy for the model. Note that the model sometimes misses the tail at the top left, e.g., for *bzip2*. However, these designs are less interesting: they have a large power increase for a very marginal performance increase. For other benchmarks (e.g., *gromacs*), we make a systematic error across all micro-architectures. However, this still leads to good relative accuracy when changing the processor configuration: the blue curve is a shifted version of the green curve. Due to the relative accuracy, the designs on the model Pareto frontier are almost exactly the same as the one on the Sniper Pareto frontier (almost all orange points are part of the blue curve). For other



Error performance:	22.9%
Error power:	15.0%
Sensitivity:	50.0%
Specificity:	92.4%
Accuracy:	82.3%
HVR:	99.0%

Error performance:	10.9%
Error power:	4.5%
Sensitivity:	70.2%
Specificity:	81.6%
Accuracy:	79.4%
HVR:	98.3%

Figure 7.5: Pareto frontiers for the *gromacs* and *xalancbmk* benchmarks.

benchmarks (e.g., *xalancbmk*), some of the orange points are off the green curve, but still close to it; these points are close to Pareto-optimal.

Next to the visual matching of the Pareto frontiers, we also show five different metrics quantifying the goodness of the Pareto frontier. We show the average absolute error on both the performance and power predictions for all designs:

$$Absolute\ error\ performance = \sum_{\forall\ designs} \frac{|P_{perf} - S_{perf}|}{S_{perf}} \tag{7.1}$$

$$Absolute\ error\ power = \sum_{\forall\ designs} \frac{|P_{power} - S_{power}|}{S_{power}} \tag{7.2}$$

Here, P is the predicted value by our model and S the simulated value taken from Sniper simulations. These metrics quantify how good our absolute predictions are for the complete design space. Figure 7.6 quantifies the absolute average errors for all SPEC CPU 2006 benchmarks. The average error for the performance and power predictions over all benchmarks are 9.3% and 4.3%, respectively. Thus, we predict performance and power well across the design space.

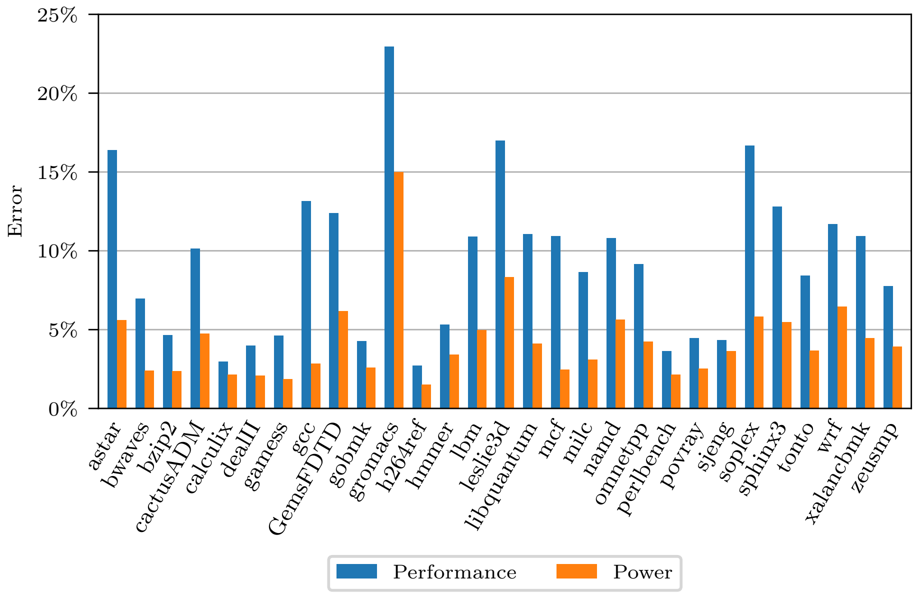


Figure 7.6: Average absolute error for performance and power predictions for the complete design space for all SPEC CPU 2006 benchmarks.

Furthermore, we also show the sensitivity, specificity and accuracy which quantify the fraction of correctly classified Pareto-optimal (true positives), non-Pareto-optimal (true negatives) and correctly classified designs in general, respectively. Note that for most predictors, there will be a trade-off between sensitivity and specificity.

$$Sensitivity = \frac{TP}{TP + FN} \tag{7.3}$$

$$Specificity = \frac{TN}{TN + FP} \tag{7.4}$$

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{7.5}$$

Here, TP, TN, FP and FN stand for true positives, true negatives, false positives and false negatives, respectively. The results for these three metrics are quantified in Figure 7.7. The average values for the sensitivity, specificity and accuracy over all benchmarks are 46.2%, 87.9%, and 76.8%, respectively.

These average values indicate that our model performs well at filtering out the non-Pareto optimal solutions (specificity), but performs not as good in detecting *all* Pareto-optimal designs (sensitivity). However, a visual inspection of the Pareto frontiers (see also Appendix C) shows that either we find only a few designs in a large cluster of Pareto-optimal designs that are very close to each other, which leads to lower sensitivity, but is also acceptable, or we miss some Pareto-optimal designs that are not useful to implement, e.g., configurations

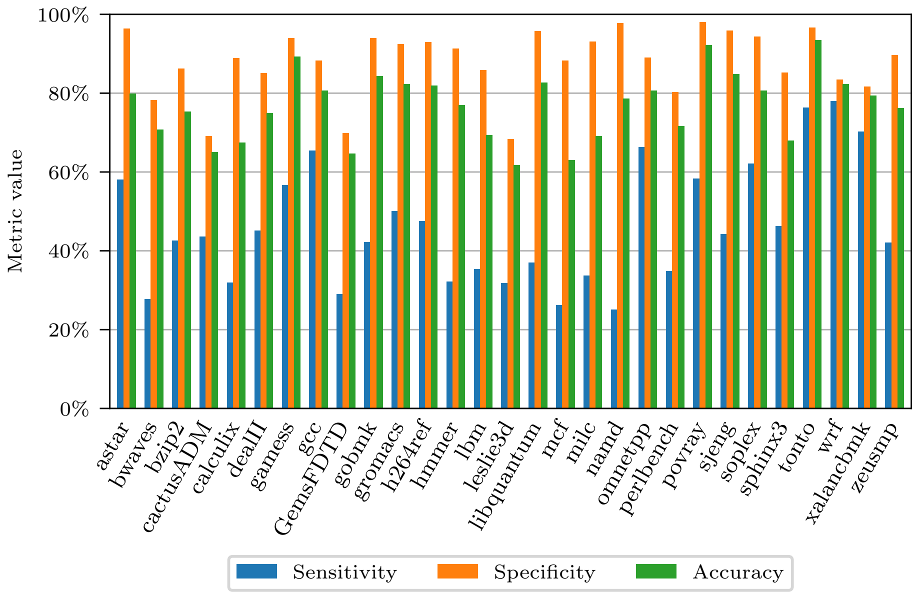


Figure 7.7: Sensitivity, specificity and accuracy of the Pareto-filtered predictions on the complete design space for all SPEC CPU 2006 benchmarks.

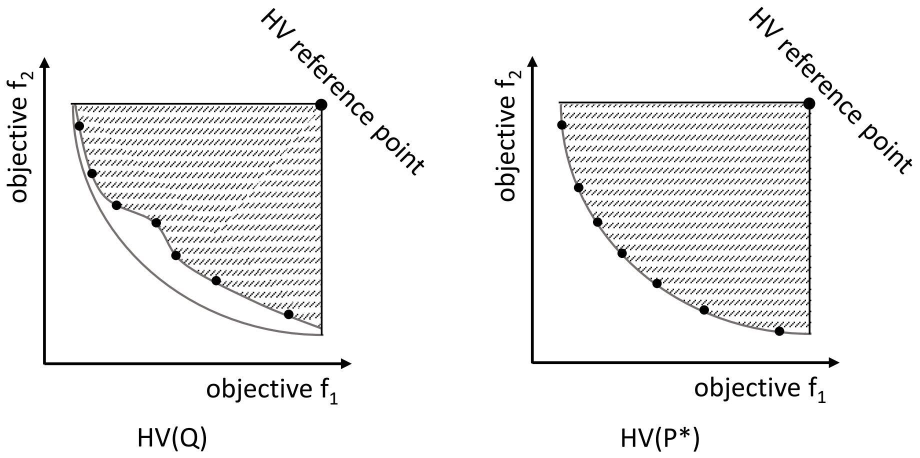


Figure 7.8: Illustration of the HVR metric calculation.

that show a large increase in power consumption while improving performance only marginally (e.g., the designs on the left vertical tail of *bzip2* that exhibit a large power increase with only a small performance gain).

The Hyper-Volume Ratio metric (HVR) [24] on the other hand shows how well we can predict the range of solutions across the entire frontier. This is important since we are interested in finding both low-power and high-performance Pareto-optimal designs at either end of the Pareto frontier. After all, sensitivity and specificity may reveal good performance, even if we only find many

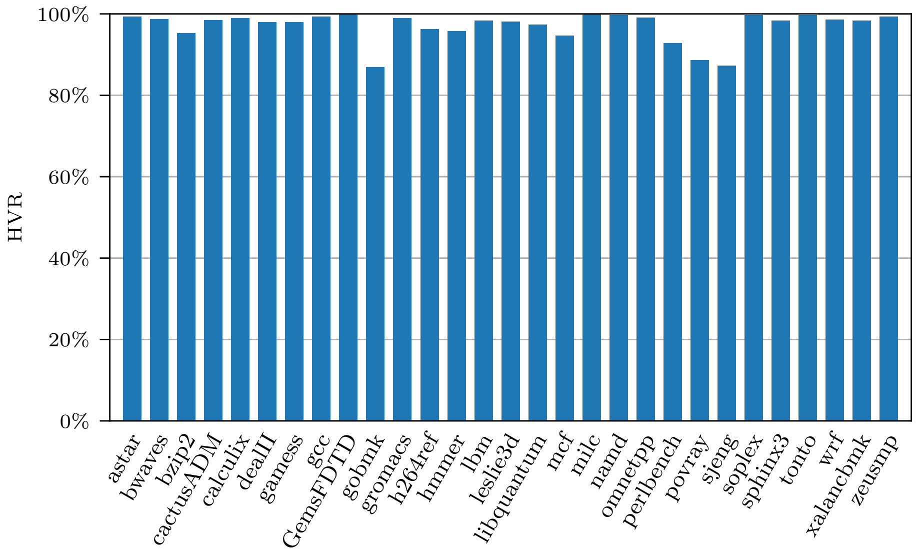


Figure 7.9: HVR of the Pareto-filtered predictions on the complete design space for all SPEC CPU 2006 benchmarks.

designs in a small range. Hence we need a metric that quantifies the range of the frontier.

According to Equation 7.7, we calculate HVR as the ratio of the hypervolume constructed by dividing the predicted Pareto-optimal configurations with simulated performance and power, $HV(Q)$, with the hypervolume formed by the Pareto front from Sniper simulations, $HV(P*)$. Figure 7.8 shows an illustration of a hypervolume for two objective functions. The left part shows the hypervolume formed by a predicted Pareto front while the right part shows the real hypervolume.

$$HV(Q) = volume(\cup_{i=1}^{Q} V_i) \tag{7.6}$$

$$HVR = \frac{HV(Q)}{HV(P*)} \tag{7.7}$$

Since our objective functions, power and performance, have different orders of magnitude, we normalize them to mitigate scaling issues and compare them to the reference point (1.1, 1.1). Note that, in our case, these equations are trivialized to a summation of the area of rectangles formed by the diagonal between the Pareto optimal points and the reference point. Figure 7.9 shows the average HVR-value for all SPEC CPU 2006 benchmarks.

The average HVR-value is 97%, indicating that our model performs well for predicting the actual range of the Pareto frontier. The only HVR-values below 90% are observed for the *gobmk*, *povray* and *sjeng* benchmarks. As shown in Appendix C, all of these benchmarks exhibit similar behavior as the *bzip2* benchmark, namely a vertical left tail indicating a steep increase in power consumption while gaining very little performance. Because the model does not predict that tail, the HVR-value is lower.

7.5 Comparison to an Empirical Model

A lot of prior work proposes predicting performance and power using empirical models [38, 43]. These models are constructed from a training set of simulated configurations. Although this training set is much smaller than the full design space, simulating the training set incurs a large overhead compared to our model, which requires only one fast profiling step per application. To quantify this overhead, we construct a polynomial regression model with polynomials up to degree 3 to predict performance and power consumption for our design space. We also evaluated higher degree polynomials, but these led to over-fitting the model. This is similar to the work of Lee et al. [43], although they used piece-wise polynomials.

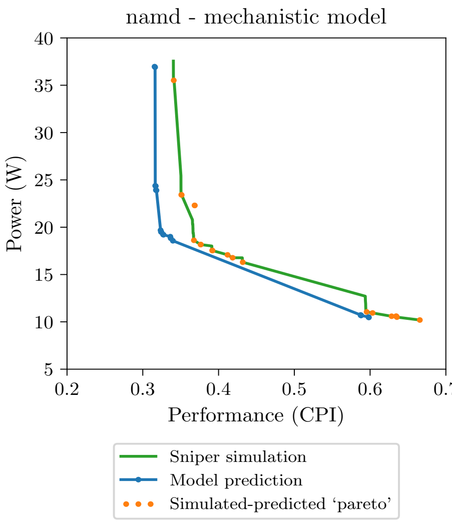
Achieving similar accuracy compared to our mechanistic model when training one model for the whole design space requires a training set of almost 2500 simulations, which lines up with the results of Lee et al. [43]. Since our profiling step is around 15× faster than simulation, and taking into account the time needed to calculate the model, this means that building a regression model is more than 250× slower than our model.

Furthermore, empirical models tend to compensate over- and under-estimations for the different parameters, resulting in less accurate predictions of the impact of changing one parameter. Therefore, the Pareto frontiers predicted using an empirical model are much less accurate than for our model, even though the average error is similar or lower for the empirical model.

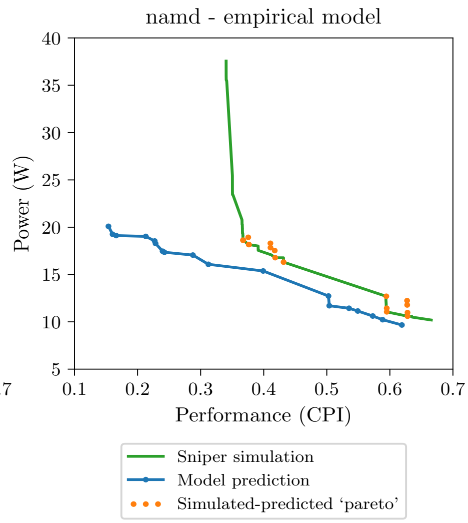
This is shown in Figure 7.10 where we show the Pareto fronts for the *namd* and *soplex* benchmarks side by side as constructed with the results of our model and the empirical technique. In the case of the *namd* benchmark, the empirical model performs worse for every statistic leading to a Pareto front that does not resemble the correct Pareto front. For the *soplex* benchmark, both the error on performance and power are lower for the empirical model. However, it performs a lot worse in actually selecting the Pareto optimal design points since the sensitivity is more than 2× lower. This can also be confirmed visually since the empirical model finds no designs in the extreme parts of the Pareto front and there are clearly a lot less orange points plotted on the green curve. Both of these cases occur for multiple other benchmarks, although there are benchmarks for which the empirical model does predict a good Pareto front.

The average prediction error for performance and power for the empirical model equals 10.0% and 4.7%, respectively, which is similar to the interval model. However, the sensitivity of the Pareto fronts generated by the empirical model is 24.3% on average, which is almost 2× lower than the average sensitivity for the Pareto-fronts found with our model (46.2%).

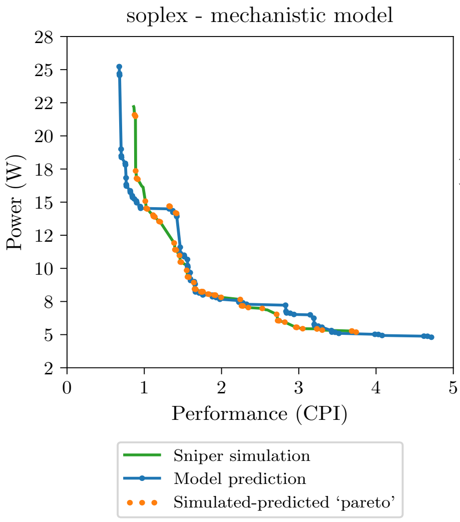
A comparison of the sensitivity values for our mechanistic model and the empirical model is shown in Figure 7.11. Although there is one benchmark, *GemsFDTD*, for which the empirical model has a significantly higher sensitivity, one can immediately see that our model outperforms the empirical model.



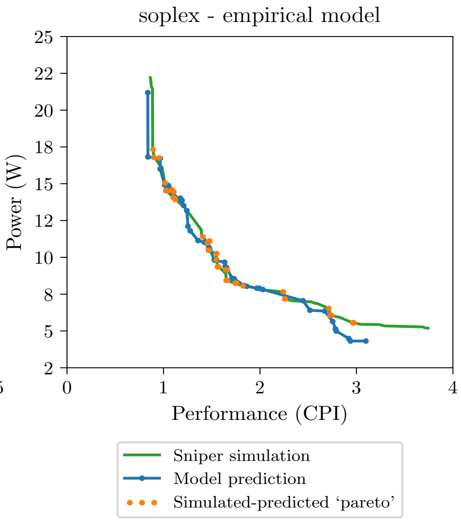
Error performance:	10.8%
Error power:	5.6%
Sensitivity:	25.0%
Specificity:	97.8%
Accuracy:	78.6%
HVR:	99.7%



Error performance:	16.0%
Error power:	4.2%
Sensitivity:	12.5%
Specificity:	95.0%
Accuracy:	73.3%
HVR:	95.3%



Error performance:	16.7%
Error power:	5.8%
Sensitivity:	62.1%
Specificity:	94.3%
Accuracy:	80.7%
HVR:	99.7%



Error performance:	6.2%
Error power:	5.5%
Sensitivity:	28.2%
Specificity:	92.1%
Accuracy:	65.0%
HVR:	98.6%

Figure 7.10: Pareto fronts for different benchmarks as calculated by our mechanistic model (left) and the empirical model (right).

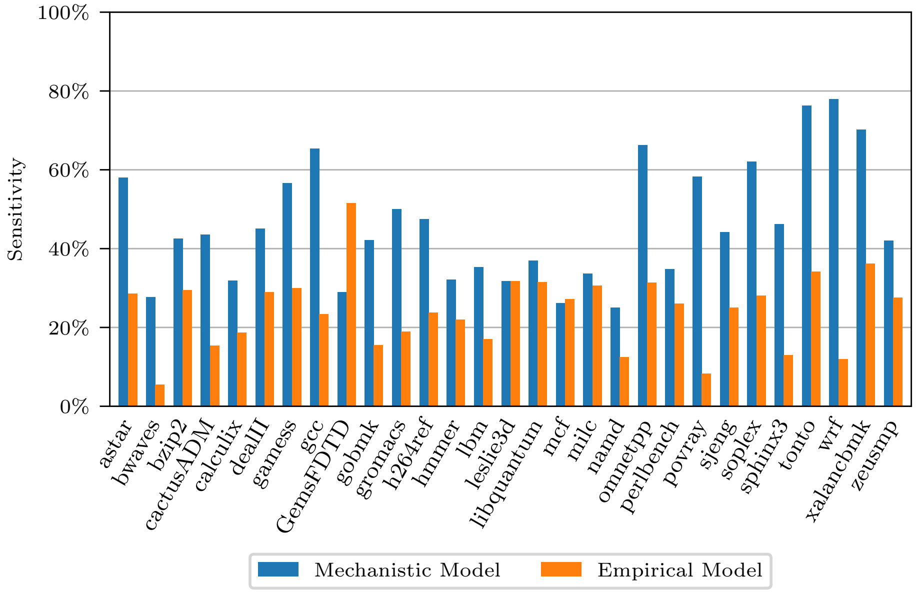


Figure 7.11: Comparison between the interval model and the mechanistic model for the sensitivity values of the Pareto-filtered predictions for the complete design space.

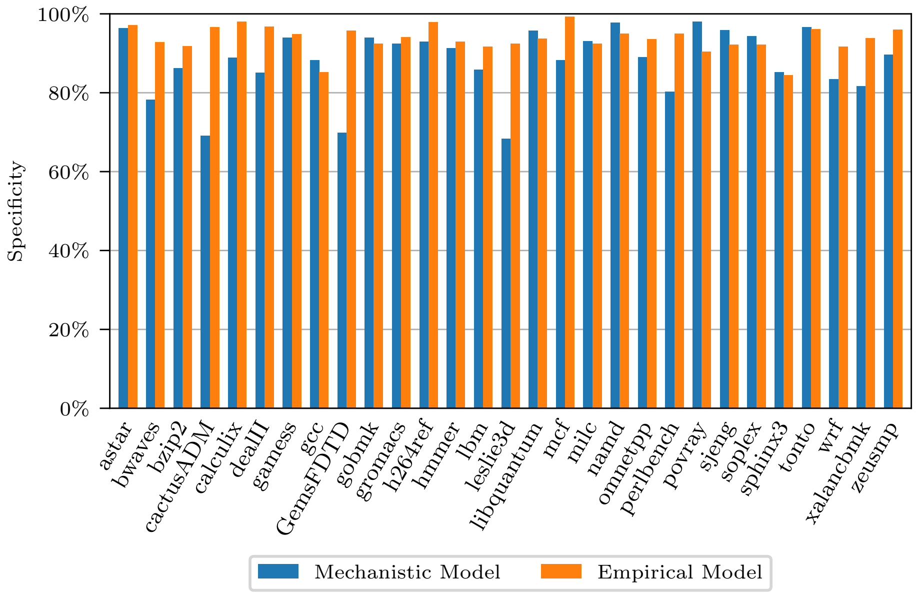


Figure 7.12: Comparison between the interval model and the mechanistic model for the specificity values of the Pareto-filtered predictions for the complete design space.

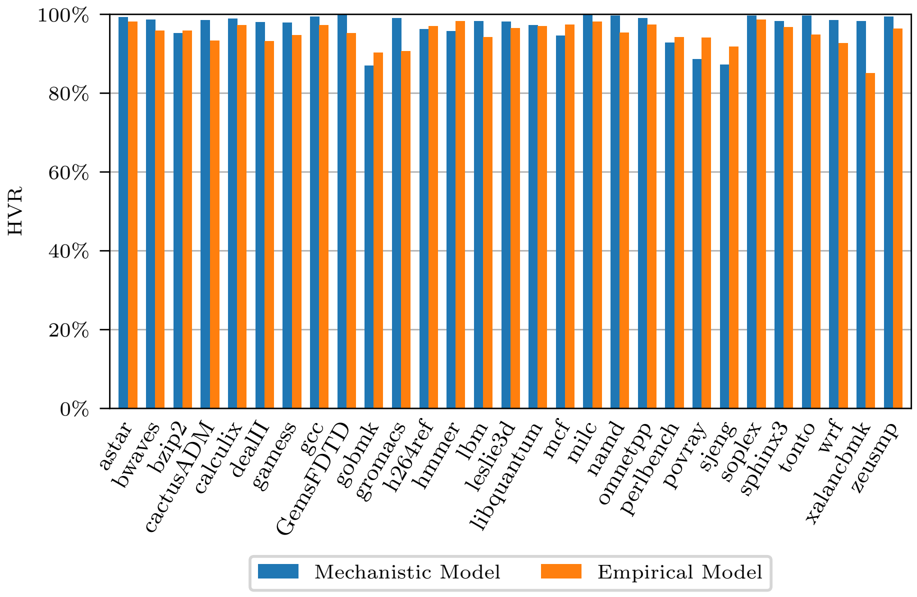


Figure 7.13: Comparison between the interval model and the mechanistic model for the HVR values of the Pareto-filtered predictions for the complete design space.

The specificity for the empirical model equals 93.6%, which is slightly higher than the 87.9% for our model. Looking at the breakdown per benchmark in Figure 7.12, we can confirm that the empirical model outperforms the mechanistic model slightly. However, this is a logical consequence of the trade-off most predictors exhibit with respect to sensitivity and specificity.

The accuracy-value is quite similar at 75.7%, while the HVR is slightly worse at 95.1%. The HVR-metric is visualized per benchmark in Figure 7.13. Hence, we can conclude that, even though the regression model performs slightly better for some statistics, our mechanistic model is better suited for design space exploration as it provides a more complete overview of the Pareto-optimal designs and is much faster.

Chapter 8

Conclusion

Your scientists were so preoccupied with whether or not they could, they didn't stop to think if they should.

– Dr. Ian Malcolm, Jurassic Park

8.1 Summary

Contemporary superscalar, out-of-order processors are incredibly complex to analyze and improve. In this thesis, we propose improvements to the interval model to more accurately model complex x86-processors and further develop it to not depend on simulation-based inputs. We also include a power prediction model using McPAT. Our main goal is to help the processor architect with tools that enable easy and fast design space exploration to develop application-specific processors to improve performance, power and energy efficiency.

Similar to other mechanistic and empirical models, our model consists of two steps, a profiling or training phase and a prediction phase. Empirical models require detailed timing-based simulations to train a model, while prior mechanistic models, as part of the profiling phase, used different types of functional simulations to collect cache miss rates, branch misprediction rates and MLP numbers. Collecting these inputs is slow and they are inherently micro-architecture dependent. We eliminate the need for those (costly) simulations by collecting application characteristics that are independent of the processor's micro-architecture. The advantage of this approach is that it results in the profiling phase becoming a one-time cost.

To speed up the profiling phase even more, we employ aggressive sampling methods. To model memory behavior, we only collect a small subset of memory accesses. We also collect small traces of instructions, called micro-traces, to extract all necessary statistics to model behavior, such as functional unit contention, within the core. We include a thorough discussion on how sampling influences the prediction error.

The prediction phase transforms the micro-architectural independent application characteristics into the inputs required by the interval model using multiple statistical models. Since the analysis phase only requires evaluating a number of statistical models and equations, it can be performed multiple orders of magnitude faster than simulation. Furthermore, when pruning a design space, our analytical model can amortize evaluation of some statistical models over different processor designs, further speeding up design space exploration.

To eliminate the need for simulation of the branch predictor, the cache hierarchy and ROB for obtaining branch mispredictions, cache miss rates and MLP, respectively, we leverage different micro-architectural independent metrics. Branch mispredictions are estimated using entropy as a metric for the (un)predictability of branches. Combined with a model for a branch predictor, the entropy of an application can be used to accurately predict the number of branch mispredictions. For predicting cache miss rates we rely on the concept of reuse distances. These reuse distances can be transformed to stack distances and mapped onto miss rates for an LRU cache hierarchy using StatStack. We extended StatStack to model a complete cache hierarchy under the assumption that the cache hierarchy consists of inclusive cache levels. In order to predict MLP, we discuss two different techniques. One of the techniques relies on cold misses to capture burstiness in the requests to the main memory. The other leverages the stride behavior static loads exhibit and combines it with information from StatStack. The stride behavior enables predicting which loads in a stream of subsequent accesses will access a new cache line and StatStack allows us to predict whether this cache line was used before and is still present in the cache hierarchy.

Next to eliminating the need for simulation-based inputs, we improve the accuracy of the core modeling for x86-based processors. Because an x86-processor uses a CISC architecture, we break down instructions into smaller micro-operations. These micro-operations are the actual unit of work that is dispatched, issued and executed. Furthermore, x86-processors are very complex, but are, depending on the application, not always perfectly balanced throughout the complete pipeline. They might have too few functional ports or functional units leading to pipeline stalls. We model this type of stalls as an effective dispatch rate equal or smaller than the physical dispatch width. We also show that, compared to the assumptions in the original interval model, branch instructions are less likely to be the last instruction of the critical path. Therefore, to calculate the branch resolution time, we introduce the average branch path as a metric for measuring the number of producing instructions leading up to a branch.

To model the memory subsystem more accurately, we introduce multiple new factors that can limit parallel processing of memory accesses. We add an extra model for MSHRs because this limits the maximum number of outstanding accesses in the cache hierarchy, which can increase the latency to fetch data.

Furthermore, while we still assume that the main memory is infinitely parallel, we take into account that transferring the data to the core takes a fixed amount of time correlated to the memory bandwidth. The memory channel between the core and the main memory can only be used by one fetch operation at a time, which incurs extra penalties.

The original interval model assumed a two-level cache hierarchy and coined that all accesses to the cache hierarchy could be hidden under other useful work which meant that only accesses to main memory could stall the processor. However, in x86-processors, a three-level cache hierarchy is more common and the last level of a three-level cache hierarchy typically has an access latency that is large enough to potentially stall the processor. This happens especially for workloads that have a large working set and that exhibit multiple last-level cache hits that depend on each other. Thus, we introduce a new penalty, called the LLC chain penalty, that models this performance loss.

One of the advantages of the stride-MLP model is that it opens up the possibility of modeling the efficacy of simple stride prefetchers. We leverage the information from the MLP model to predict which loads can be prefetched, and whether they can be prefetched in a timely manner. In addition, we show that one cannot ignore the effect of these prefetchers when trying to achieve good accuracy.

Next to easy and fast design space exploration, we also want to offer insight into application behavior because this can help in optimization of applications and/or processors. We enable this through building CPI stacks to gain insight into where the cycles go during an application's execution. We use these to show how the model can be used to pinpoint the exact cause of performance loss and propose a better performing processor architecture. This is a definitive advantage over empirical or black-box models which can only predict the metric for which they are trained and can thus offer little insight in performance losses.

The introduction of micro-architecture independent application characteristics and their transformation to the inputs required by the interval model, introduces the potential for additional prediction errors. We include a study detailing the contribution of each micro-architecture independent component to the total performance prediction error. This study shows that the prediction of the memory components contributes most to the additional prediction error.

Evaluation of the model shows that we achieve an absolute prediction error of 9.3% for performance and 4.3% for power compared to a cycle-level accurate simulator. These accuracy numbers are obtained across a large design space where the sizes of the most important micro-architectural structures are modified. Evaluation of this design space can be performed 300× faster compared to using cycle-level accurate simulation. Furthermore, we show that despite the prediction error on our model, we achieve good relative accuracy. This is the most important feature of our model as it enables design space exploration where multiple processor designs can be compared to each other accurately. We also show that this important property does not necessarily hold for regression-based machine learning techniques which rely on averaging out predictions across the design space.

8.2 Future Work

This thesis showed the possibility of predicting performance and power for an application running on a processor using only micro-architectural independent characteristics. However, the current work is subject to a number of constraints. These constraints could be further relaxed in future work.

8.2.1 Multi-core Processors

The first constraint of this work is that it focuses on single-core processors executing only one application. Contemporary processors usually feature multiple different cores in one chip [53] and can execute multiple applications concurrently. Depending on the micro-architecture, those multi-core processors share multiple levels of the cache hierarchy and the bus for accessing main memory. The sharing of these resources introduces extra penalties. For example, one application can evict data from a shared cache that another application still needs or one application has to wait until another releases the bus to main memory.

To accurately model performance the sharing of the cache hierarchy and bus and the attributed performance loss has to be included in the interval model. A first-order assumption could be that both are divided equally over the different applications. Hence, if two applications are co-executing, both would have access to half of the total cache size and half of the DRAM bandwidth [18]. More complex techniques such as described in StatCC [27] could improve the accuracy of these first-order approximations. Furthermore, as the resource sharing by the co-running applications influences the respective execution times, an iterative approach to predict the final performance might be required [67]. Depending on how sensitive the co-running applications are to the sharing of the cache, bus and DRAM resources, it might be required to adopt more complex modeling techniques.

8.2.2 SMT Processors

Simultaneous MultiThreaded (SMT) processors [64] are processors in which a core can execute multiple threads at the same time. The granularity on which these threads execute concurrently can differ, but essentially this type of processors share most core structures over different threads. The aim of this type of processor is to improve the utilization of processor resources. The extensive sharing of all core resources influences the performance of the processor significantly from the viewpoint of the co-running applications. Extending the current interval model to include SMT processors would entail little work on the profiler side of our framework, but would necessitate rethinking most modeling steps. For example, can the sharing of the functional units be modeled as simply averaging the instruction mixes of different threads? What is the influence of the (possibly dynamic) ROB sharing on the MLP? Do we need to include models for contention in the load-store queue? Hence, this extension does require a significant research effort. Note that contemporary processors often combine multiple cores with SMT.

8.2.3 Multithreaded Workloads

The set of applications used in this work perform all of their work using a single thread. However, partially due to the advent of multi-core processors, a new set of applications started to gain traction, namely multithreaded workloads. This type of workloads are an important subset of all current applications and uses multiple threads to calculate a solution to a problem. Usually the work that these applications have to perform is divided among the threads, but it is possible that the threads have to operate on the same data. If there is a possibility of the threads modifying a data element simultaneously, the programmer has to supply a mechanism that guarantees the threads modify that element sequentially. Otherwise, race conditions could lead to undefined behavior and/or incorrect program execution. Often used synchronization mechanisms are critical sections, locks, barriers, etc. All of these mechanisms add extra overhead to the program's execution time depending on, among others, the degree of sharing and the amount of threads. Thus this extra overhead also needs to be modeled. Modeling these types of workloads requires updates to both the profiling and modeling tools and is definitely not trivial. We are currently looking into the modeling of multithreaded workloads [23].

8.2.4 Different ISAs

The current implementation of our profiling tool relies on Intel's Pin [48] for obtaining application characteristics. Inherently, this constraints the usage of our profiling and modeling to x86-based processors. While these processors are by far the most prevalent in contemporary desktops and servers, mobile devices often feature an ARM processor. These ARM processors use the ARM ISA which cannot be instrumented using Pin. Therefore, in order to model performance of an ARM processor, a different instrumentation framework is needed. One such possible framework is DynamoRIO [2].

Another approach could be to collect application statistics independently of the ISA using an intermediary representation of the application similar to the work by Shao et al. [57]. One possible framework to collect those statistics would be LLVM [7].

Both of these extensions require a significant amount of engineering work and research. During my internship at ARM, I did extend the models described in this thesis for the ARM architecture. Unfortunately, there were some outliers which I was unable to fix in the short time span of the internship. The reason for these outliers are likely caused by ARM-processors being more resource-constrained, which proved to be more challenging to capture in the analytical model than anticipated. However, I strongly believe that there are no fundamental limitations as to why the analytical model cannot be extended and fine-tuned to more accurately model ARM processors. Furthermore, because the developed models include confidential IP, they are not open source.

Appendix A

CPI stacks

Appendix A shows CPI stacks for all SPEC CPU 2006 benchmarks for our reference architecture. This is possible because the model is a summation of different penalty terms, see Equation A.1. Building CPI stacks is one of the most useful features of the interval model.

$$C = \frac{N}{D_{\mathit{eff}}} + m_{bpred}(c_{res} + c_{fe}) + \sum_i m_{ILi} c_{Li+1} + \frac{m_{LLC}(c_{mem} + c_{bus})}{MLP} + P_{hLLC} \quad \text{(A.1)}$$

$$D_{\mathit{eff}} = \min\left(D, \frac{ROB}{lat \cdot CP(ROB)}, \frac{N}{N_p}, \frac{N \cdot U_i}{N_i}, \frac{N \cdot U_j}{N_j \cdot lat_j}\right) \quad \text{(A.2)}$$

To gain more insight, we split the base component into four different sub-components. These sub-components follow the minimizing operation calculating the effective dispatch rate, D_{eff}, based on the physical dispatch width, inter-instruction dependences and contention in the functional ports and units as shown in Equation A.2. Hence, if the physical dispatch width is the minimum and thus the most limiting factor, only the base component is visible. If there are stricter limits due to dependences, issue ports or functional units, this is shown as an extra stack on top. To calculate this extra stack, we divide the number of micro-operations by the respective dispatch rate and subtract the preceding components as shown in the below equations:

$$Base = \frac{N}{D_{physical}} \quad \text{(A.3)}$$

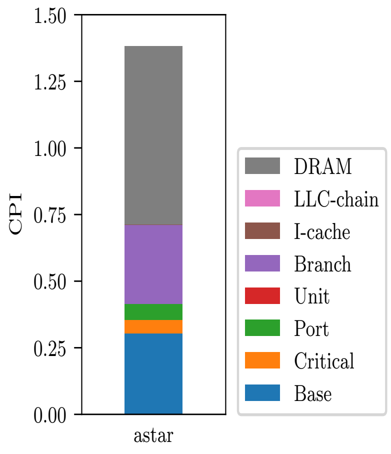
$$Crit = \max\left(0, \frac{N}{D_{critical}} - Base\right) \quad \text{(A.4)}$$

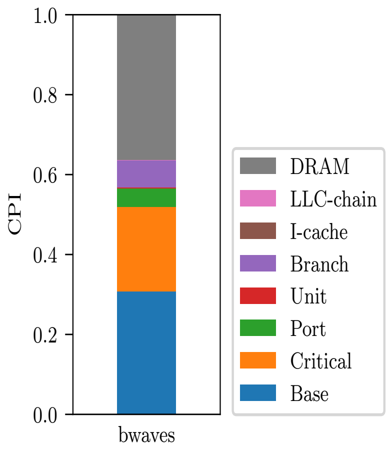
$$Port = \max\left(0, \frac{N}{D_{port}} - Critical - Base\right) \quad \text{(A.5)}$$

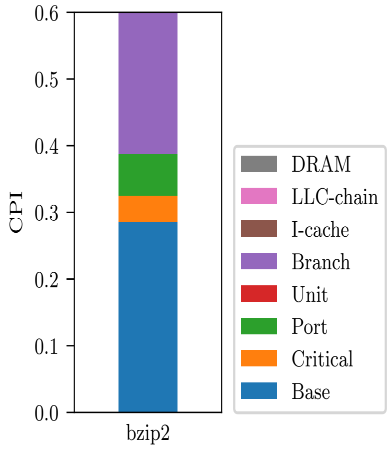
$$Unit = \max\left(0, \frac{N}{D_{unit}} - Port - Critical - Base\right) \quad \text{(A.6)}$$

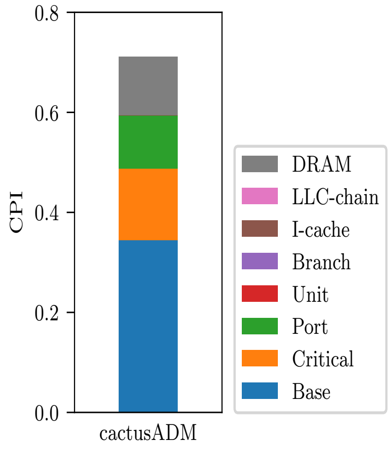
The eight different components (for which the first four components are the base sub-components) visualized in the CPI stacks are described below:

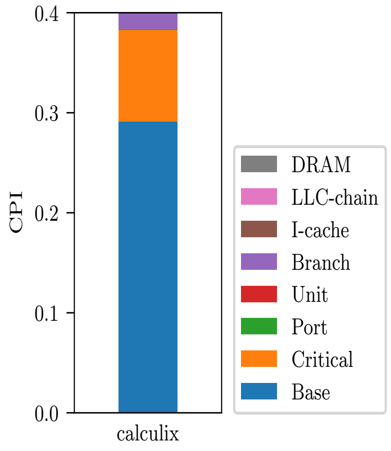
- Base component (calculated following Equation A.3): The minimum execution time equals the number of micro-operations divided by the physical dispatch width.
- Critical component (calculated following Equation A.4): The penalty introduced due to long dependence chains which lower the processor's issue and execution rate.
- Port component (calculated following Equation A.5): The penalty introduced because multiple functional units are connected to the same issue port, thus requiring multiple cycles to issue all micro-operations.
- Unit component (calculated following Equation A.6): An extra penalty for instructions that have to be executed by non-pipelined functional units.
- Branch component (second term in Equation A.1): The penalty for the number of branch mispredictions multiplied by the sum of the front-end refill time and the branch resolution time.
- I-cache component (third term in Equation A.1): The penalty due to the number of I-cache misses times the latency.
- DRAM component (fourth term in Equation A.1): The penalty for the number of LLC misses times their respective latency and divided by the MLP.
- LLC-chain component (fifth term in Equation A.1): The penalty component indicating that the ROB blocks due to multiple LLC hits on the same dependence path.

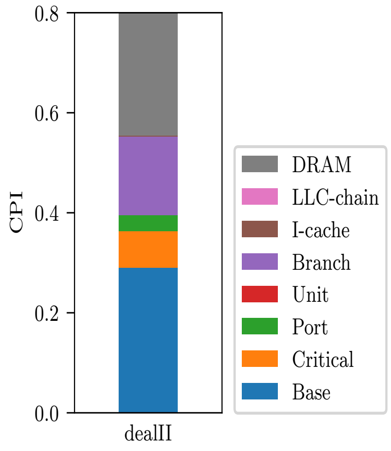


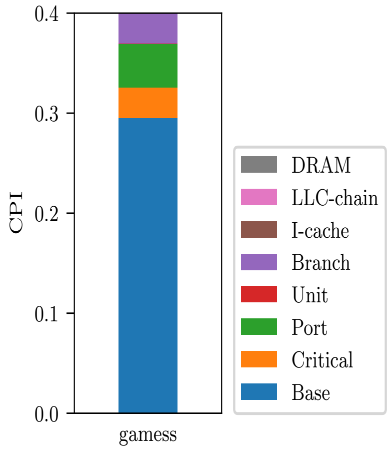


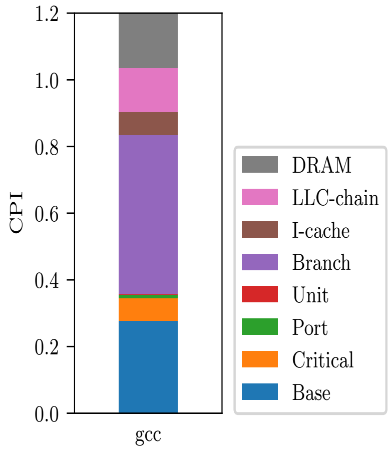

0.6
0.5
0.4
0.3
0.2
0.1
0.0
CPI
bzip2
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

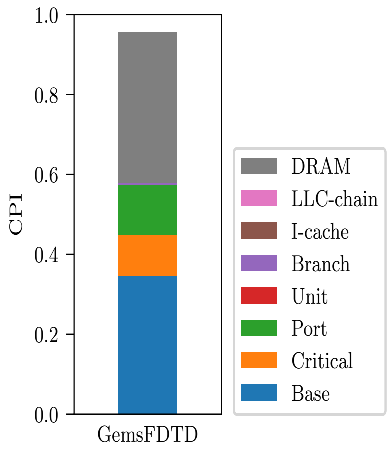

0.8
0.6
0.4
0.2
0.0
CPI
cactusADM
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

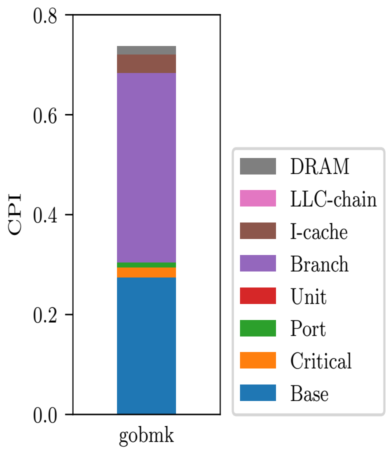

0.4
0.3
0.2
0.1
0.0
CPI
calculix
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

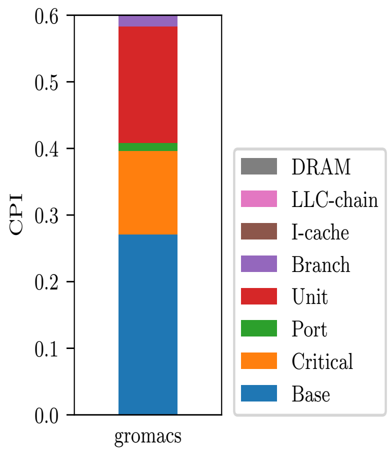

0.8
0.6
0.4
0.2
0.0
CPI
dealII
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

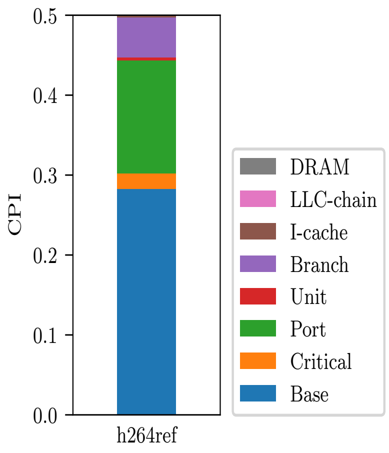

0.4
0.3
0.2
0.1
0.0
CPI
gamess
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

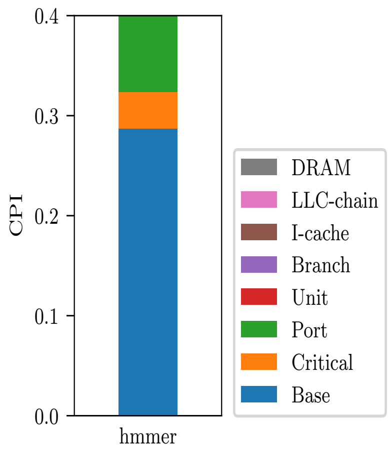

1.2
1.0
0.8
0.6
0.4
0.2
0.0
CPI
gcc
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

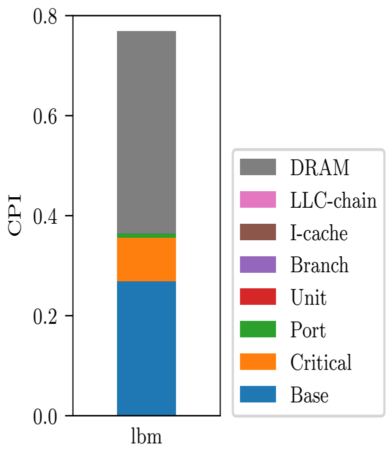

CPI
1.0
0.8
0.6
0.4
0.2
0.0
GemsFDTD
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

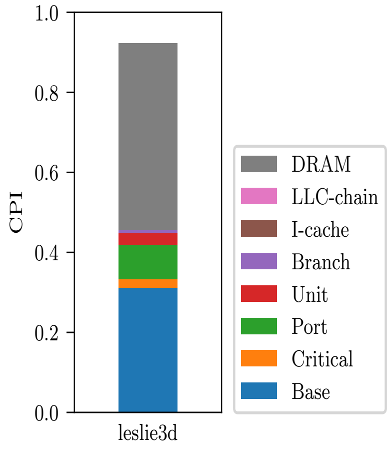

CPI
0.8
0.6
0.4
0.2
0.0
gobmk
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

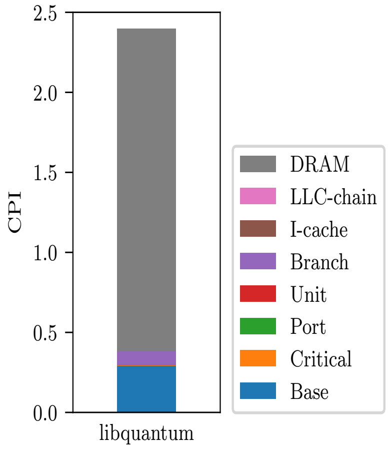

CPI
0.6
0.5
0.4
0.3
0.2
0.1
0.0
gromacs
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

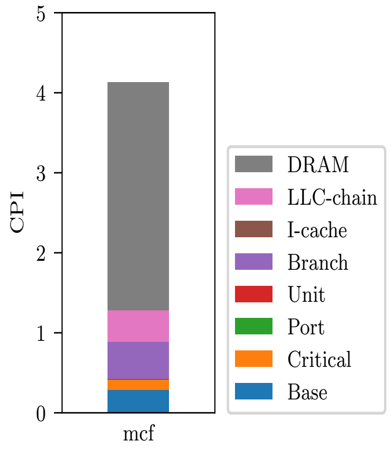

CPI
0.5
0.4
0.3
0.2
0.1
0.0
h264ref
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

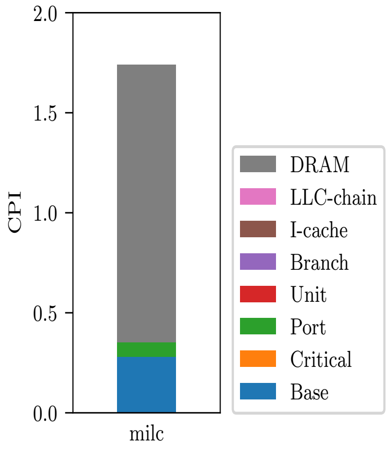

CPI
0.4
0.3
0.2
0.1
0.0
hmmer
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

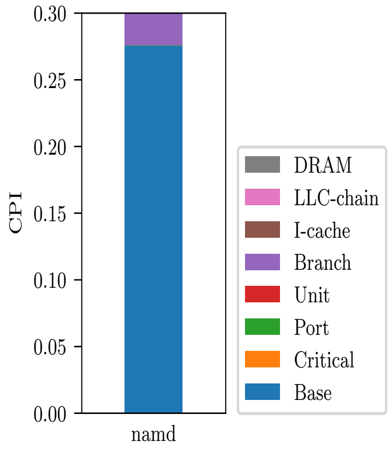

CPI
0.8
0.6
0.4
0.2
0.0
lbm
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

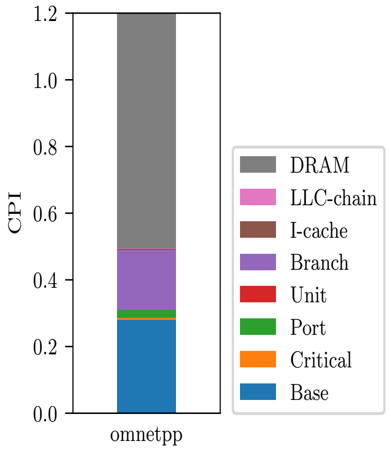

CPI
0.0
0.2
0.4
0.6
0.8
1.0
leslie3d
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

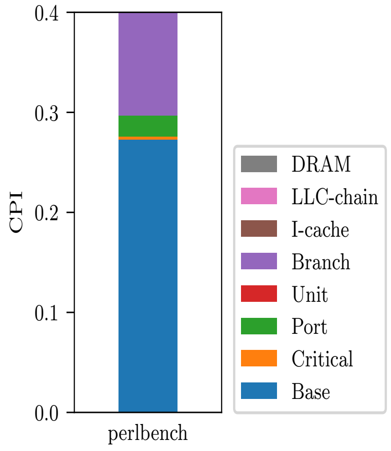

CPI
0.0
0.5
1.0
1.5
2.0
2.5
libquantum
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

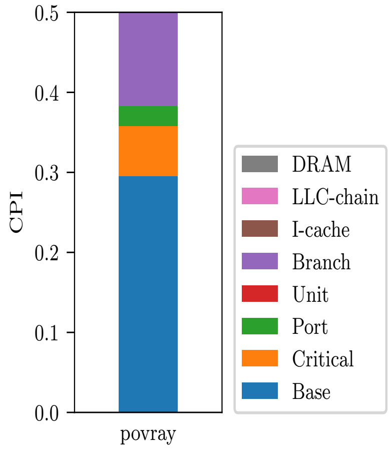

CPI
0
1
2
3
4
5
mcf
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

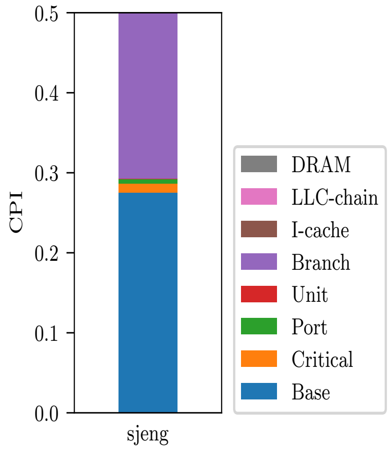

CPI
0.0
0.5
1.0
1.5
2.0
milc
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

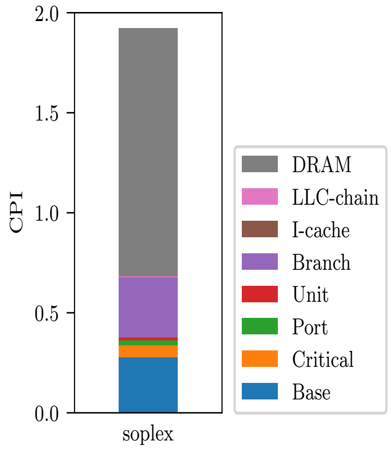

CPI
0.00
0.05
0.10
0.15
0.20
0.25
0.30
namd
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

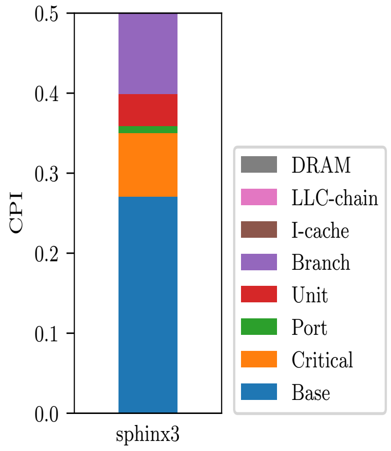

CPI
0.0
0.2
0.4
0.6
0.8
1.0
1.2
omnetpp
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

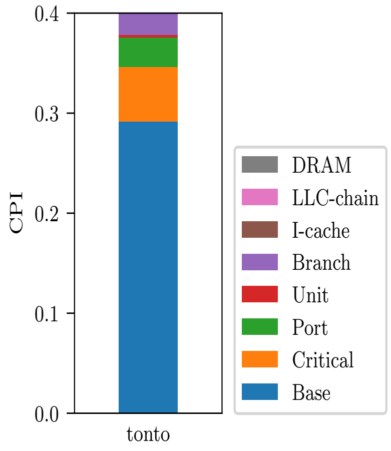

0.0
0.1
0.2
0.3
0.4
CPI
perlbench
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

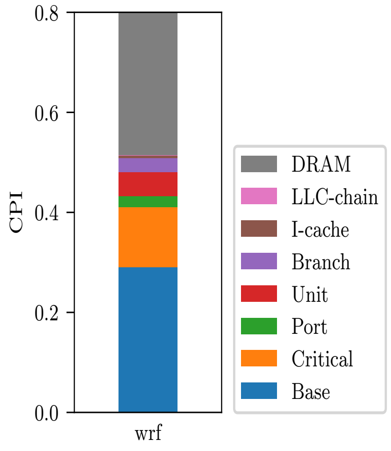

0.0
0.1
0.2
0.3
0.4
0.5
CPI
povray
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base


0.0
0.1
0.2
0.3
0.4
0.5
CPI
sjeng
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

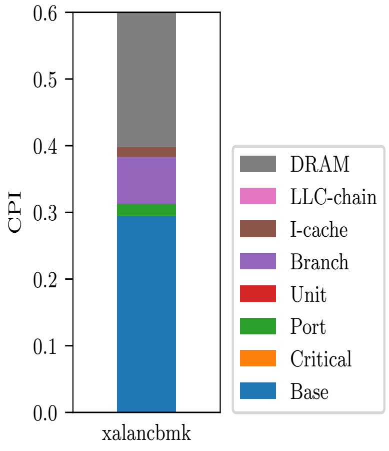

0.0
0.5
1.0
1.5
2.0
CPI
soplex
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base


0.0
0.1
0.2
0.3
0.4
0.5
CPI
sphinx3
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

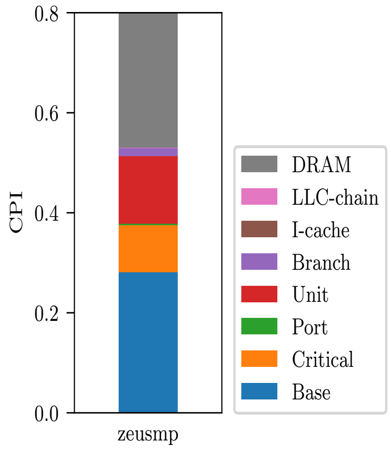

0.0
0.1
0.2
0.3
0.4
CPI
tonto
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base



CPI
0.0
0.2
0.4
0.6
0.8
wrf
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base



CPI
0.0
0.1
0.2
0.3
0.4
0.5
0.6
xalancbmk
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base



CPI
0.0
0.2
0.4
0.6
0.8
zeusmp
DRAM
LLC-chain
I-cache
Branch
Unit
Port
Critical
Base

Appendix B

Phase plots

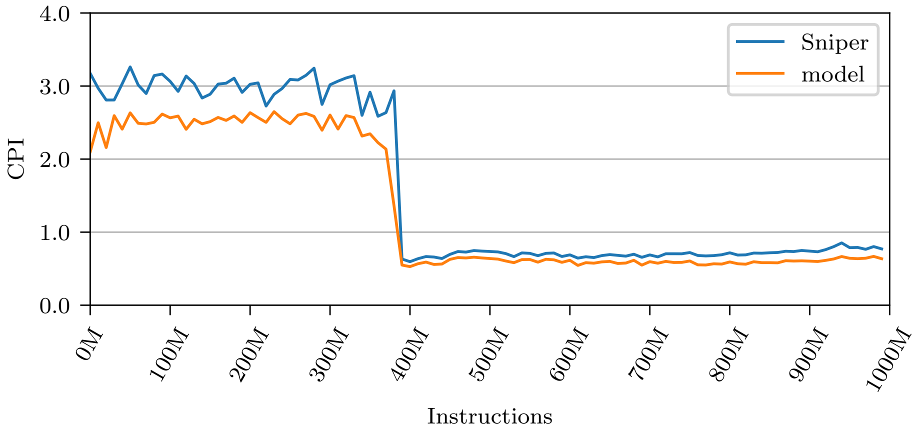
Appendix B shows graphs detailing the CPI over time as estimated by the model versus Sniper. Under each figure, we put the benchmark name, its average error measured over the whole execution, and the phase accuracy coefficient (PAC). Together these metrics give a good idea of the accuracy. The PAC is a measure for the average error on the phase behavior and is calculated as follows:

$$PAC = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{C_S(i-1) - C_S(i)}{\overline{C_S}} - \frac{C_M(i-1) - C_M(i)}{\overline{C_M}}\right| \tag{B.1}$$

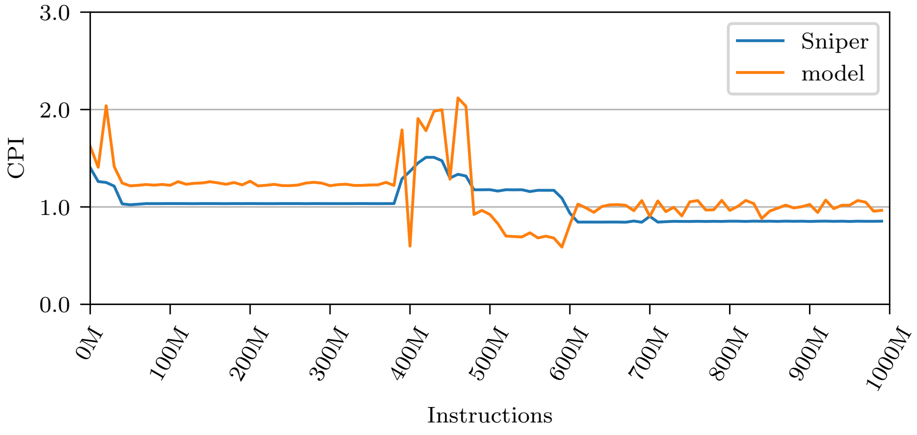
We include the reason for inaccurately estimating some applications' phase behavior:

- *gromacs*: This benchmark performs a lot of divisions and square root operations. Both of these instructions are sent to the non-pipelined division unit. This leads to severe unit contention at very small time scales, which we cannot track accurately because we average over at least 1K instructions. PAC is lower compared to the overall Average error, because even though the model does not accurately predict the severity of the contention, it does predict the relatively constant CPI most of the time, and it also accurately predicts the timing of the performance dips.
- *lbm*: The first half of the execution contains mostly cold misses, and the second half mostly conflict misses. We use probabilities over all loads, leading to an underestimation of the number of misses that depend on one another. Hence, we overestimate MLP and underestimate CPI for the first half. For the second half, we cannot estimate where exactly conflict misses occur in a window and have to divide them uniformly leading to an underestimation of the MLP and an overestimation of the CPI.
- *libquantum*: In the second half we are unable to estimate some CPI peaks because we do not accurately predict the number of cache misses, leading to an underestimation of the DRAM component.

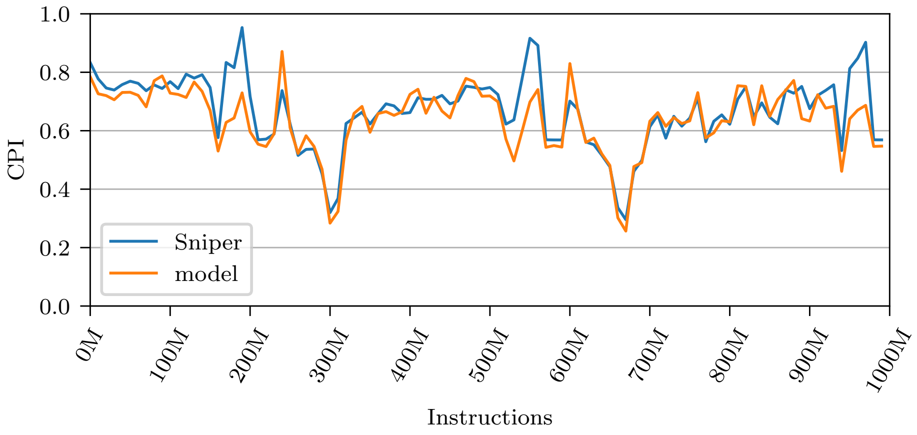
- *milc*: In the second half of the execution, we are unable to estimate some of the lower CPI phases, because we cannot estimate the bursty behavior of the conflict misses. This leads to an underestimation of the MLP and an overestimation of the CPI.
- *soplex*: In-between the CPI peaks, we underestimate the MLP (and thus overestimate CPI) because here, the conflict misses occur in bursts, while we assume that they occur uniformly. At the CPI peaks, there is limited MLP due to the fact that most LLC misses are on the same dependence path; the model again assumes that all misses are uniformly distributed across all dependence paths, leading to a higher MLP and a lower estimated CPI. PAC is worse compared to the overall Average error: the overall Average error is somewhat reduced by compensating over- and underestimations, which is not the case for the PAC metric.



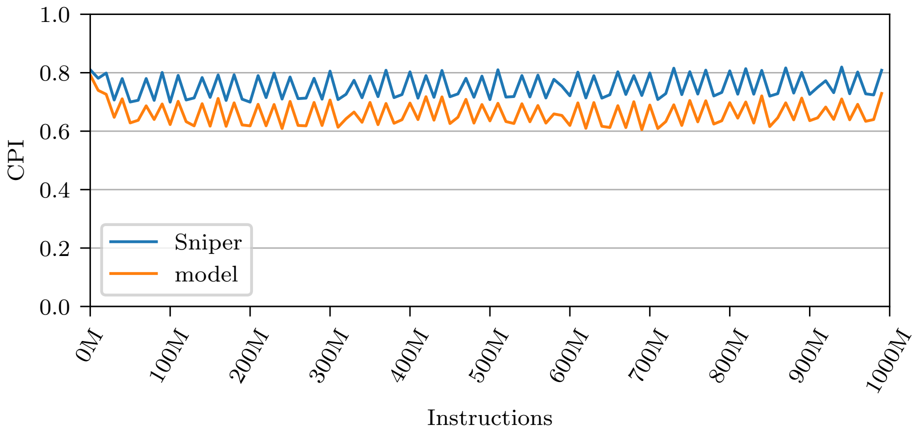
astar – error: -16.9% – PAC: 6.2%



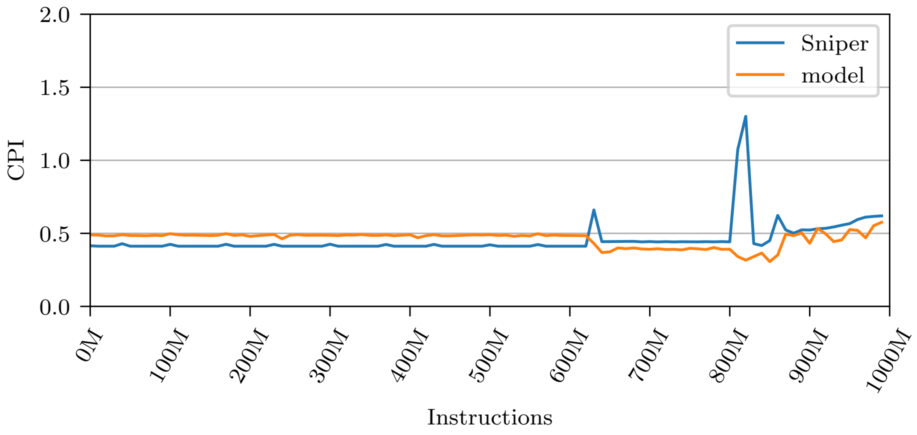
bwaves – error: 11.8% – PAC: 9.7%



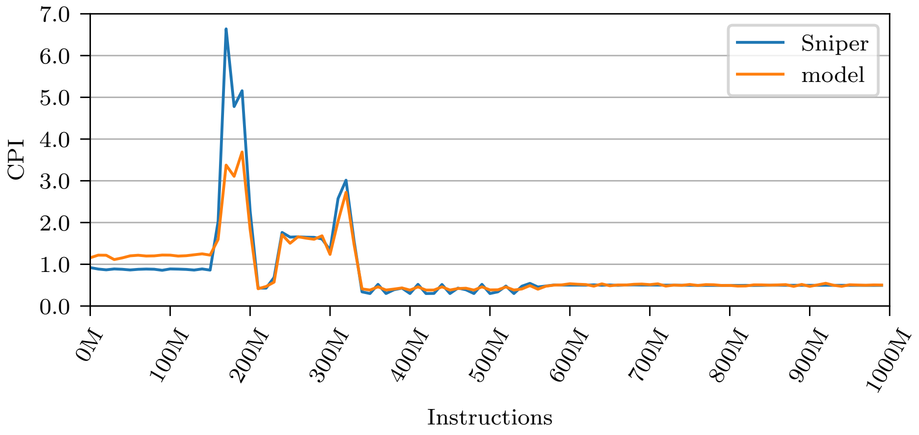
bzip2 – error: -4.3% – PAC: 6.7%



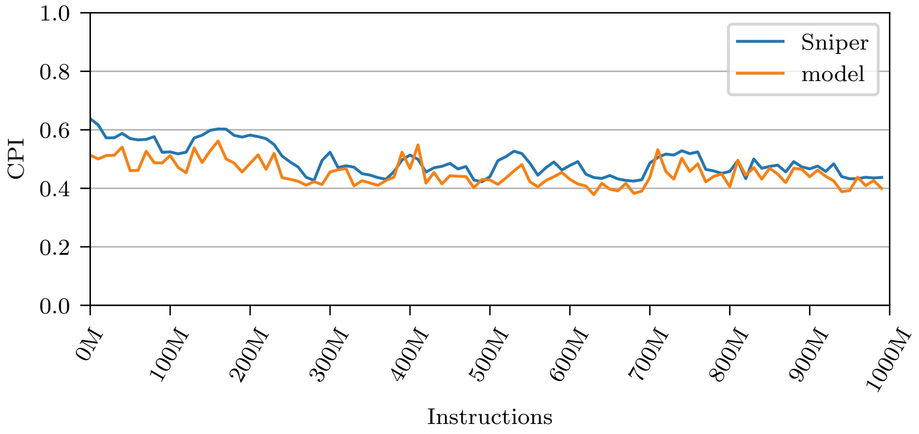
cactusADM – error: -12.2% – PAC: 1.9%



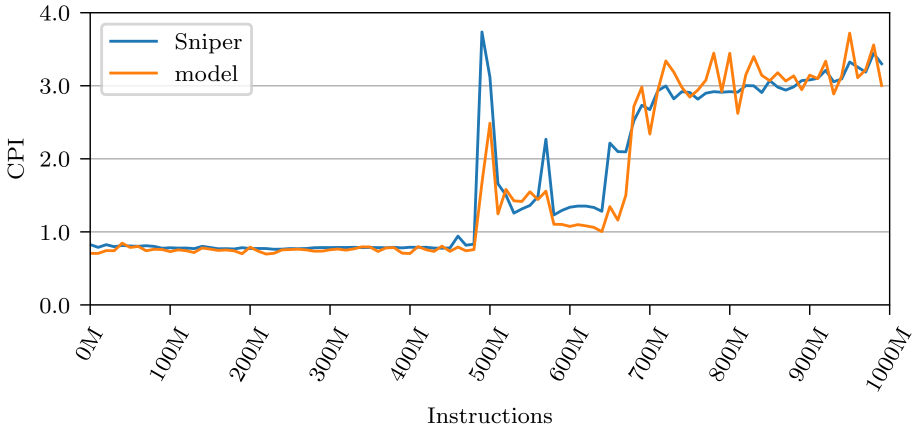
calculix – error: 0.9% – PAC: 8.2%



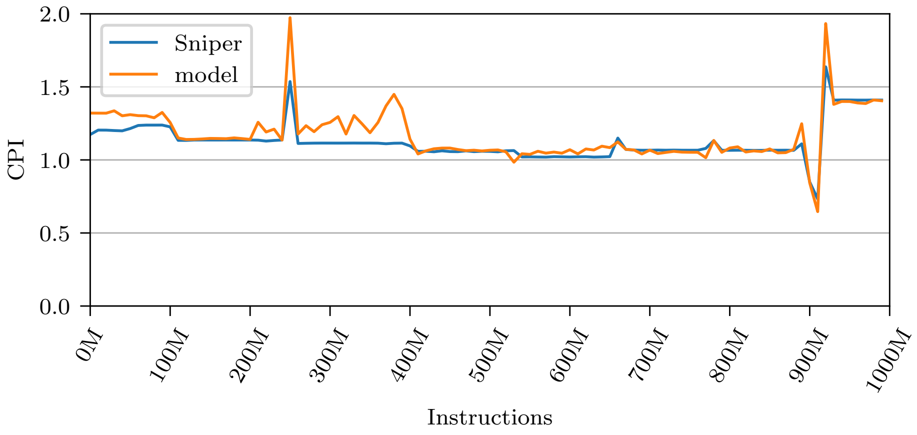
dealII – error: -3.2% – PAC: 14.1%



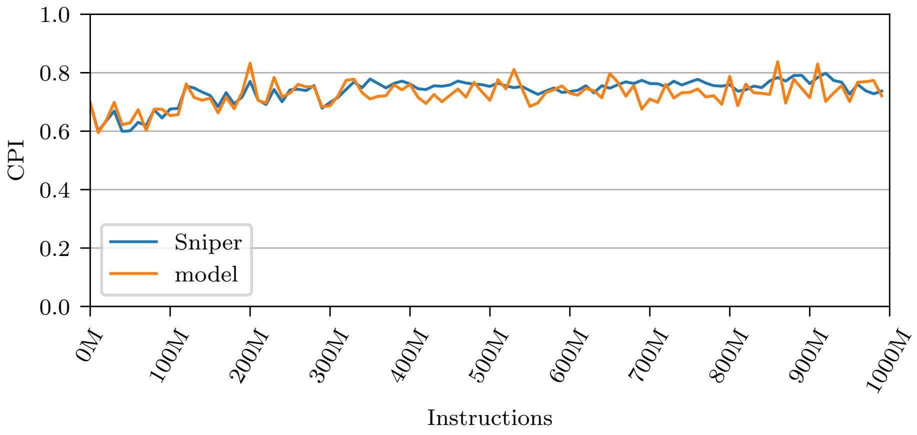
gamess – error: -8.6% – PAC: 6.5%



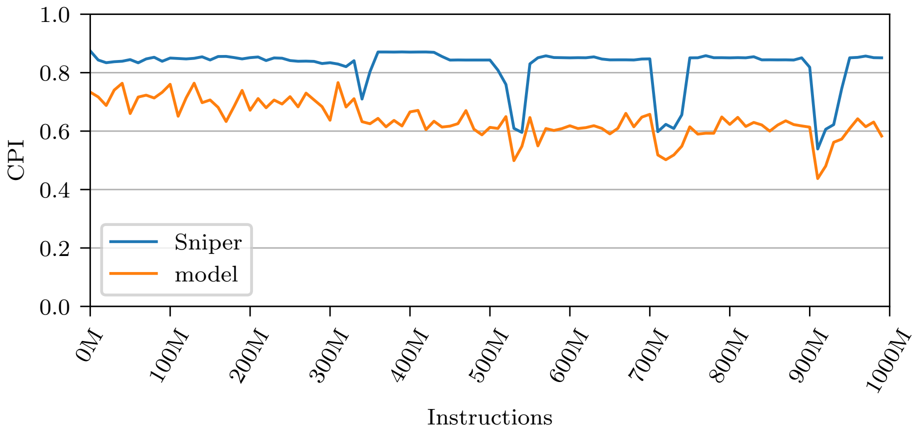
gcc – error: -3.6% – PAC: 10.8%



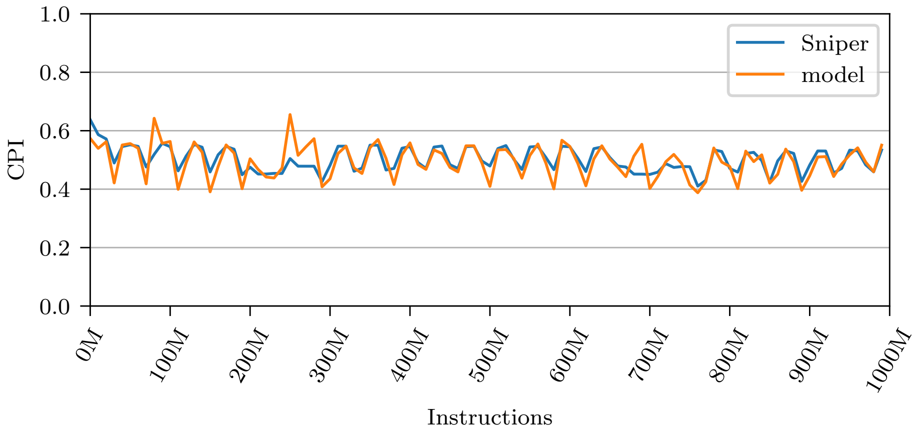
GemsFDTD – error: 4.0% – PAC: 3.9%



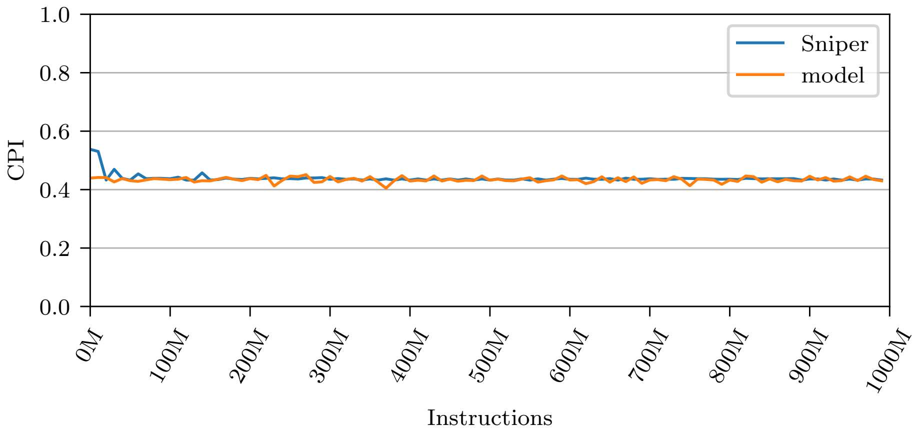
gobmk – error: -1.7% – PAC: 4.6%



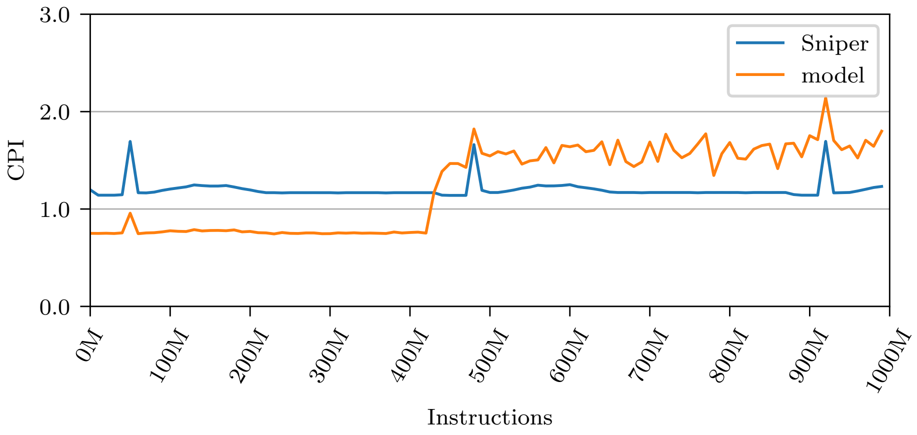
gromacs – error: -22.3% – PAC: 5.5%



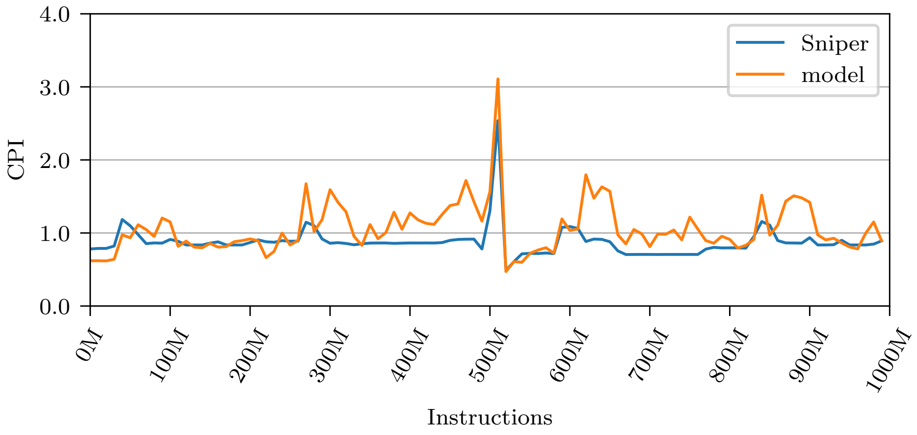
h264ref – error: -1.3% – PAC: 7.4%



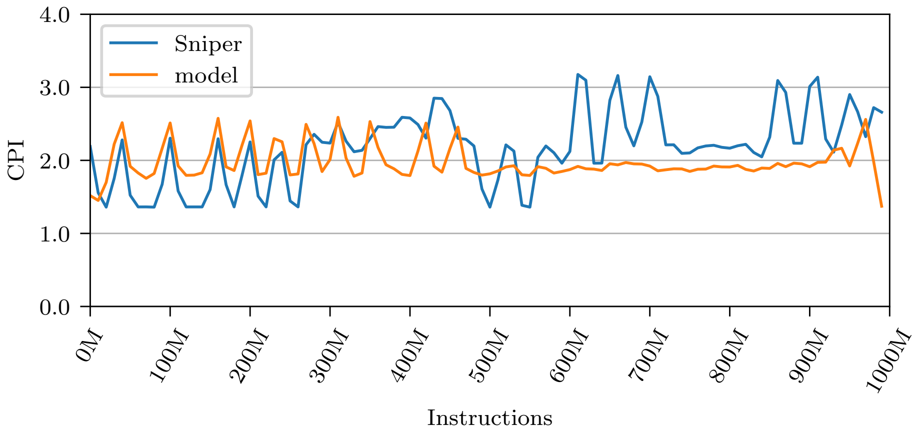
hmmer – error: -1.1% – PAC: 2.8%



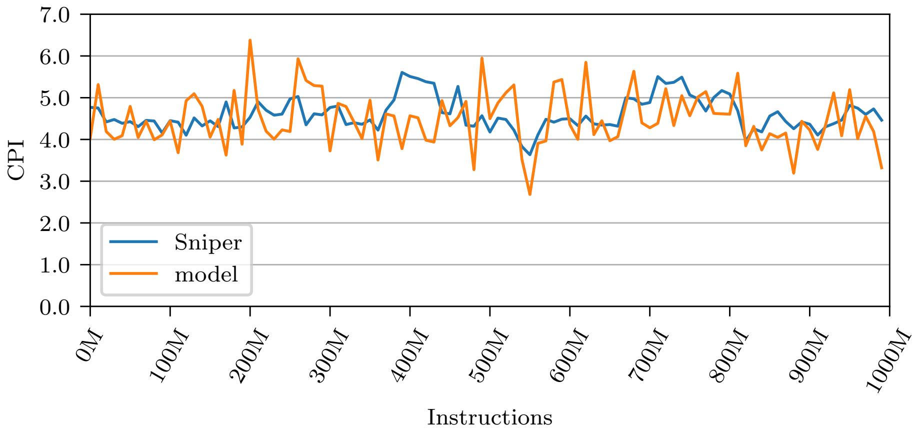
lbm – error: 3.3% – PAC: 6.5%



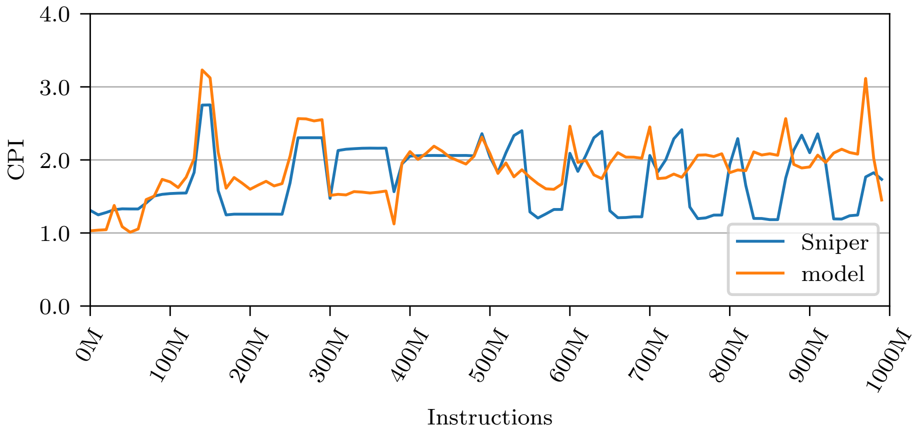
leslie3d – error: 20.7% – PAC: 15.1%



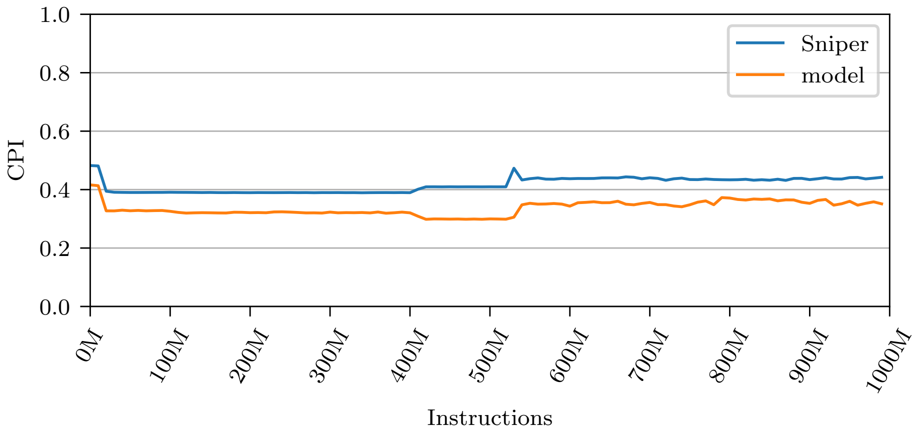
libquantum – error: -9.1% – PAC: 13.6%



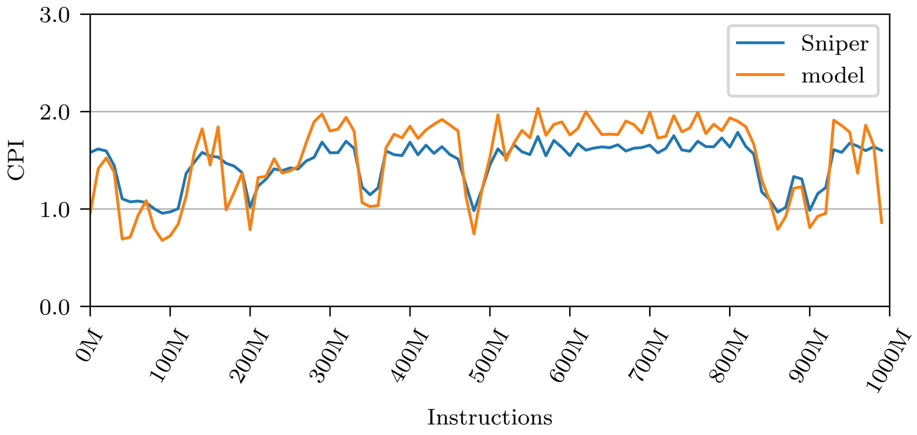
mcf – error: -2.8% – PAC: 15.6%



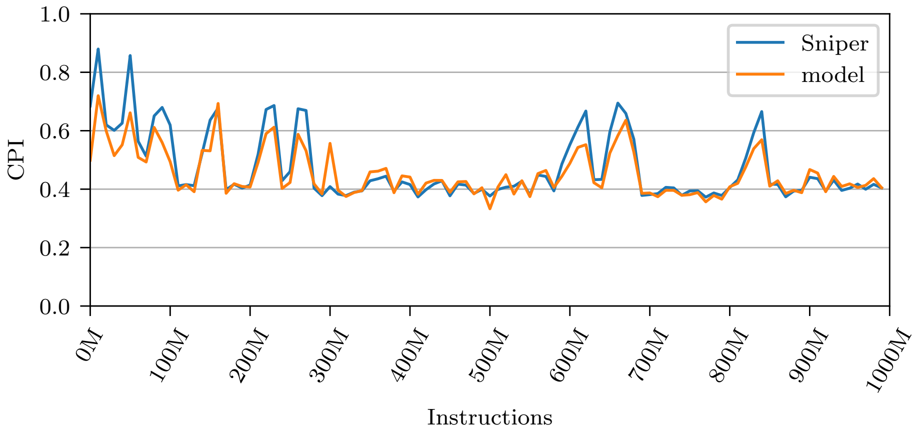
milc – error: 8.8% – PAC: 13.6%



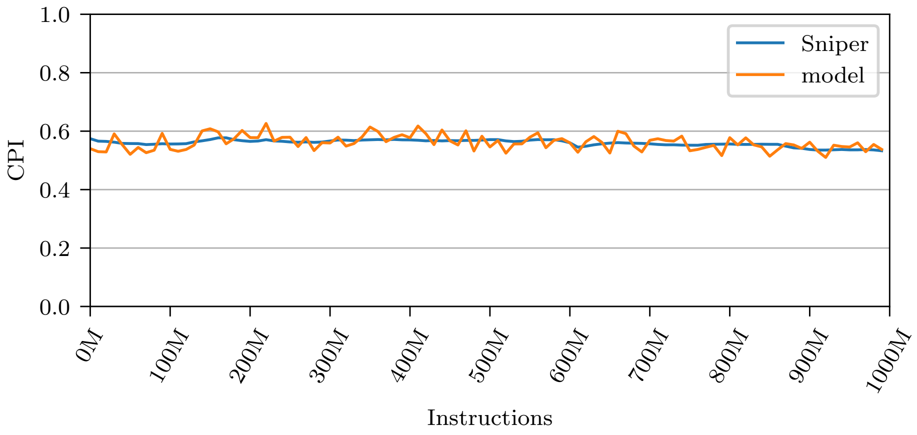
namd – error: -19.1% – PAC: 1.5%



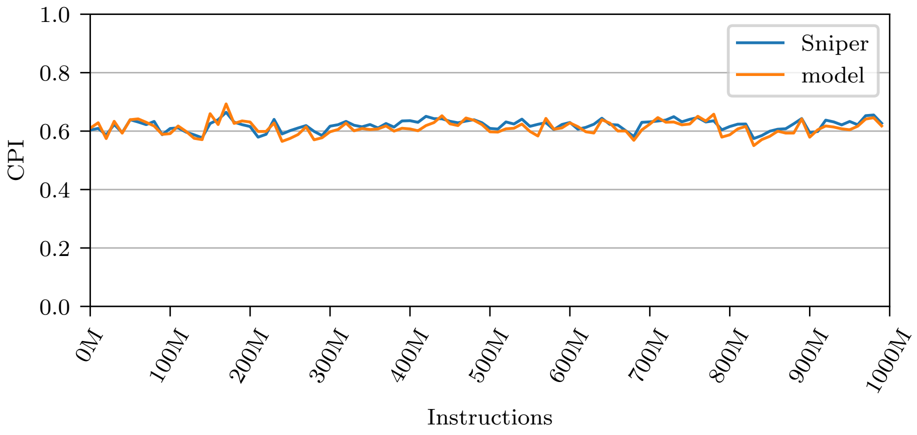
omnetpp – error: 3.5% – PAC: 9.8%



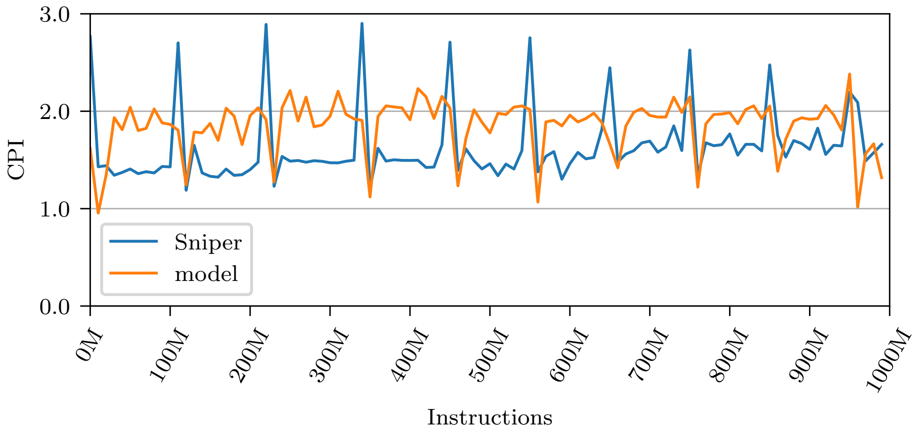
perlbench – error: -4.2% – PAC: 6.9%



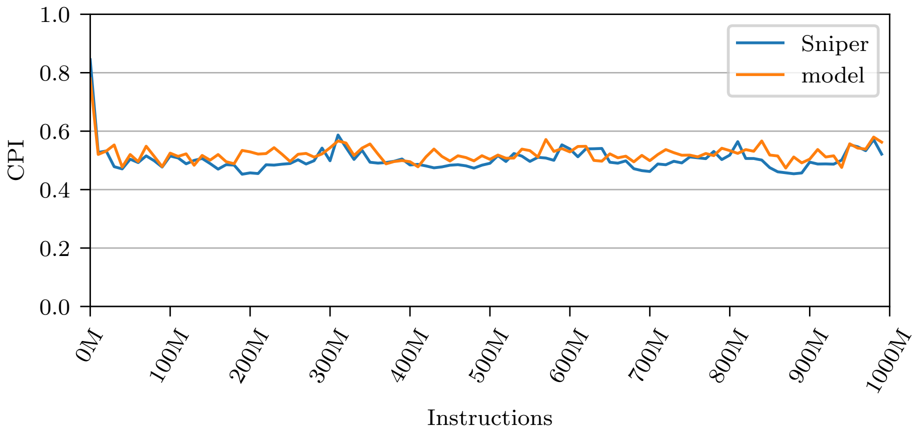
povray – error: 0.3% – PAC: 4.4%



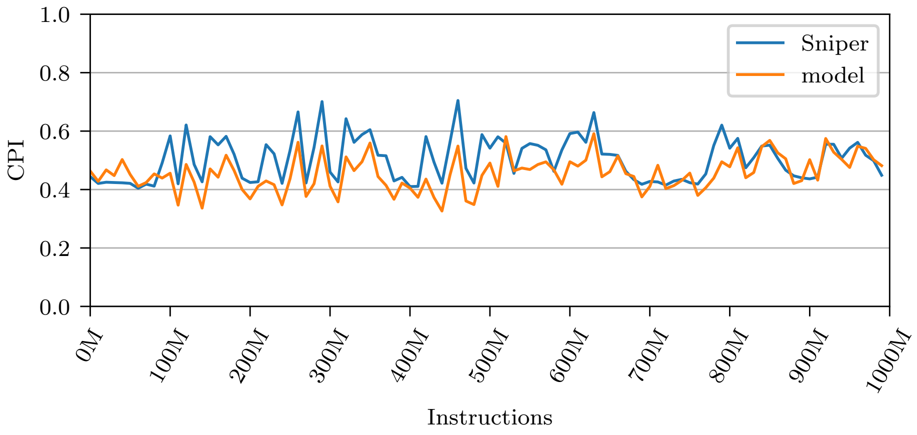
sjeng – error: -1.3% – PAC: 2.3%



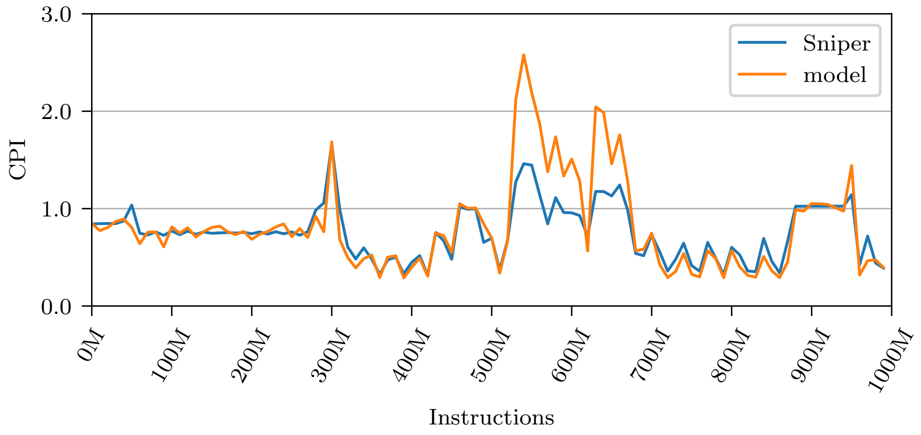
soplex – error: 13.6% – PAC: 18.7%



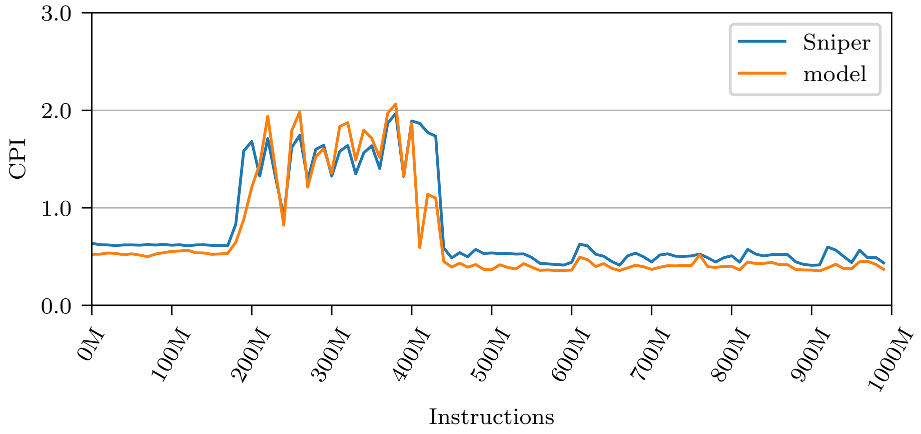
sphinx3 – error: 3.8% – PAC: 5.0%



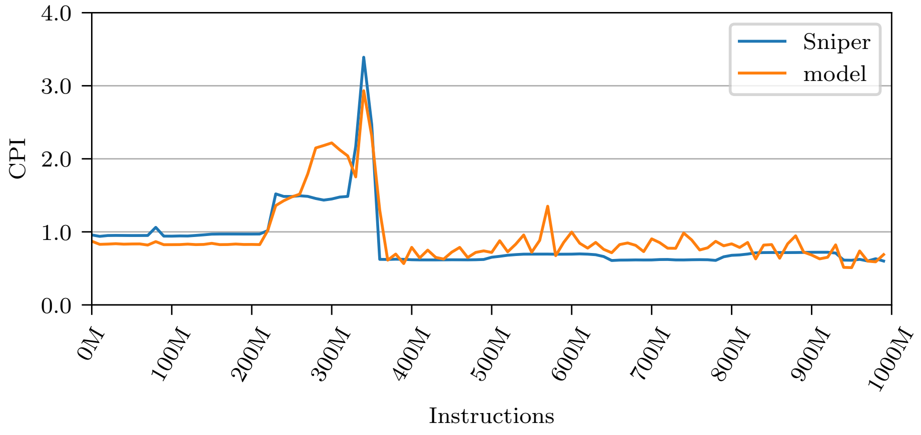
tonto – error: -9.5% – PAC: 8.2%



wrf – error: 8.0% – PAC: 14.4%



xalancbmk – error: -11.6% – PAC: 10.9%



zeusmp – error: 8.7% – PAC: 13.3%

Appendix C

Pareto plots

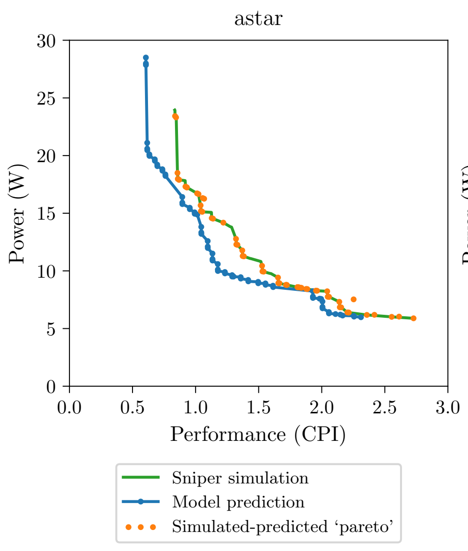
Appendix C contains all Pareto frontiers for the SPEC CPU 2006 benchmarks as calculated by the model (blue curve) and simulated by Sniper (green curve). The orange points are the configurations for which our model predicted they are Pareto-optimal, but shown with their simulated performance and power consumption. The difference between the blue and green curves shows the error of the model, while the difference between the orange points and the green curve indicates how well we can predict actual Pareto-optimal configurations.

Next to the visual matching, we show various metrics underneath each figure: the average absolute error for performance and power, as well as sensitivity, specificity and the Hypervolume Ratio (HVR) [24]. Sensitivity and specificity quantify the fraction of predicted actual Pareto-optimal and non-Pareto-optimal designs, respectively. HVR quantifies how well we can predict the range of solutions across the entire frontier. Put together, these metrics denote how good each predicted Pareto frontier is.

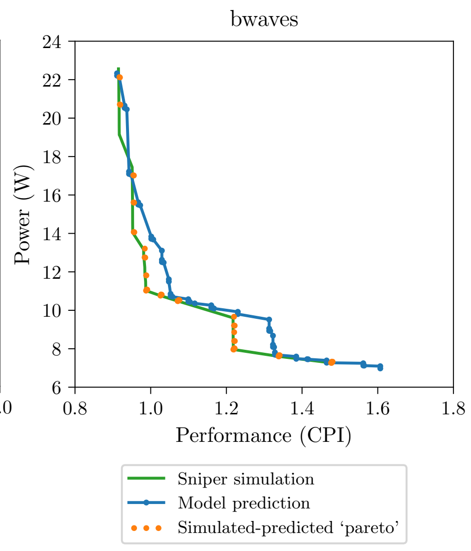
The average values over the whole design space are 9.3%, 4.3%, 46.2%, 87.9%, 76.8% and 97.0% for the error on performance and power, specificity, sensitivity, accuracy and HVR respectively. Hence, our model is very good at predicting the actual range of the Pareto frontier (HVR) and also at filtering out the non-Pareto optimal solutions (sensitivity), but performs less good on detecting *all* Pareto-optimal designs (specificity). However, a visual inspection of the Pareto frontiers shows us that either we find only a few designs in a large cluster of Pareto-optimal designs that are very close to each other, which leads to lower sensitivity — but which we deem acceptable — or we miss some Pareto-optimal designs that are not useful to implement (e.g., the designs on the left vertical tail of `bzip2`: large power increase with only a small performance gain).

We include additional explanation for some of the Pareto frontiers:

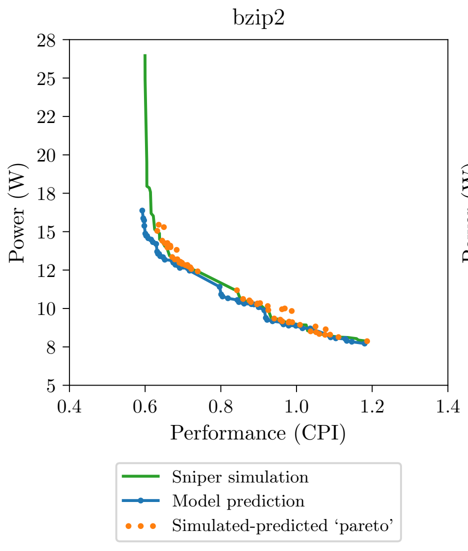
- `bzip2`, `h264ref`, `gobmk` and `soplex`: The model misses the top-left tail of the Pareto frontier, which appears to be almost vertical. However, these designs are less interesting to find because they represent a marginal increase in performance while power increases substantially. Furthermore, this tail is always comprised of less than 5% of the total designs.
- `gromacs`: As shown in the phase graph (see Appendix A), we make a systematic error across all configurations. However, this still leads to good relative accuracy when changing the processor configuration. This systematic error is shown in the Pareto frontier: the blue curve is a shifted version of the green curve where the error for all CPI values is indeed around 22% to the left. Due to the good relative accuracy, the designs on the model Pareto frontier are almost exactly the same as the ones on the Sniper Pareto frontier (almost all orange points are part of the green curve).
- `hmmer`: There is a tail on the right that we do not predict accurately. This is similar to `bzip2` etc. for which we not accurately predict the tail on the left. We still see most of the Pareto-optimal designs on that right tail, but not all Pareto-optimal designs in the knees of the curve, leading to lower sensitivity.
- `perlbench`: We do not predict the left vertical tail, and we predict two dense clusters of Pareto-optimal designs. However, this is not an issue since Sniper also does not select Pareto optimal designs in between those clusters. The predicted frontier correctly connects the Pareto-optimal clusters.
- `sjeng`: The model does not find any of the designs on the left vertical tail because it cannot properly estimate the decrease in branch misprediction rate of using the gshare16 branch predictor. The model classifies all branch predictors as performing approximately the same, while in fact, the gshare16 branch predictor outperforms the others for the larger dispatch widths.
- `sphinx3`: We do not see the left vertical tail, which in this case is actually built up out around 20 designs. However, those designs are all clustered on 4 places on the vertical tail, and are less interesting because they double power consumption for a gain in performance of less than 5%.
- `xalancbmk`: Here we observe designs which are not actually Pareto-optimal. However, these points are still close to being Pareto-optimal.



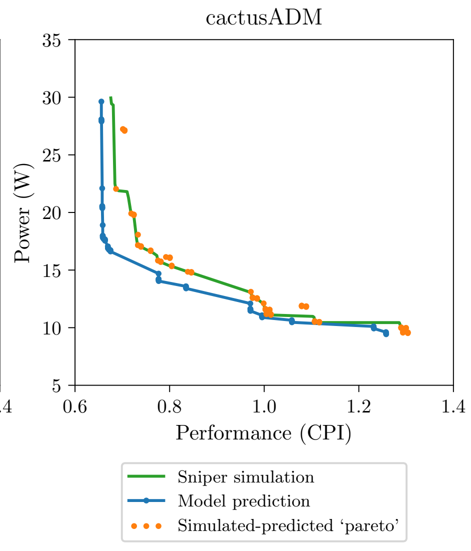
Error performance:	16.4%
Error power:	5.6%
Sensitivity:	58.1%
Specificity:	96.4%
Accuracy:	79.8%
HVR:	99.3%



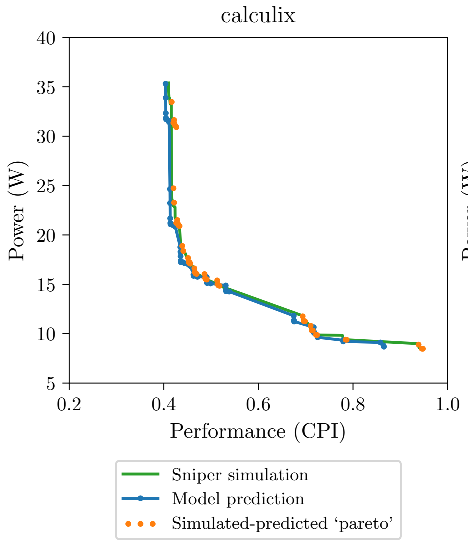
Error performance:	7.0%
Error power:	2.4%
Sensitivity:	27.8%
Specificity:	78.3%
Accuracy:	70.8%
HVR:	98.6%



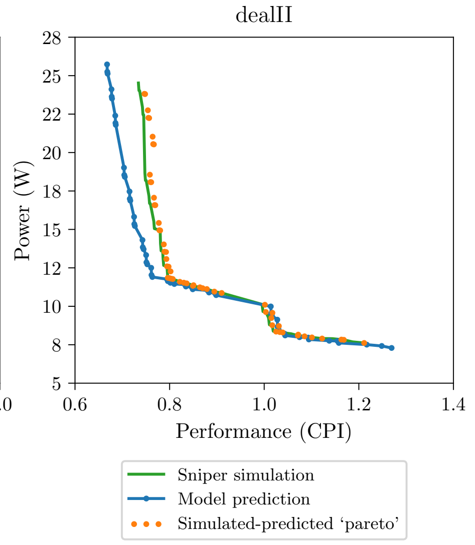
Error performance:	4.6%
Error power:	2.4%
Sensitivity:	42.6%
Specificity:	86.3%
Accuracy:	75.3%
HVR:	95.3%



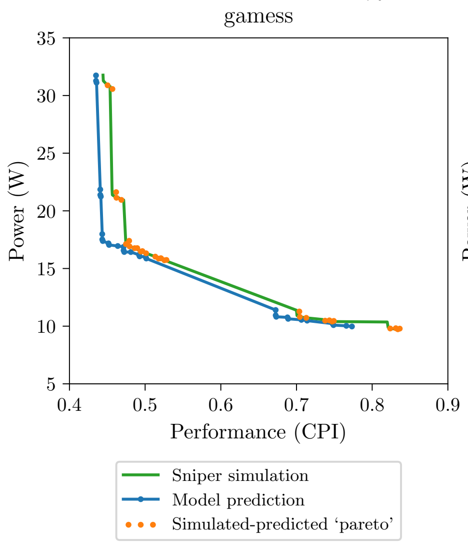
Error performance	10.1%
Error power:	4.8%
Sensitivity:	43.6%
Specificity:	69.1%
Accuracy:	65.0%
HVR:	98.5%



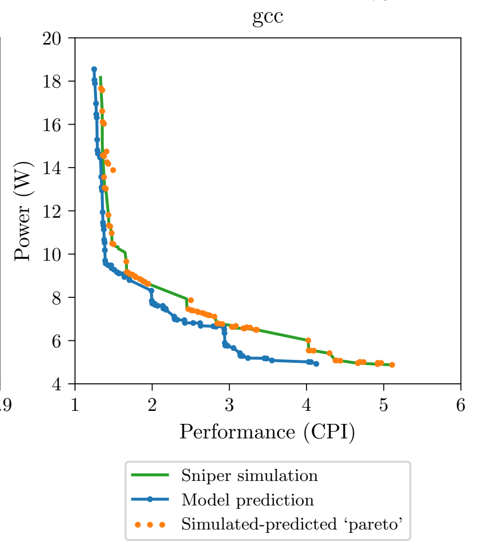
Error performance:	3.0%
Error power:	2.1%
Sensitivity:	31.9%
Specificity:	88.8%
Accuracy:	67.5%
HVR:	99.0%



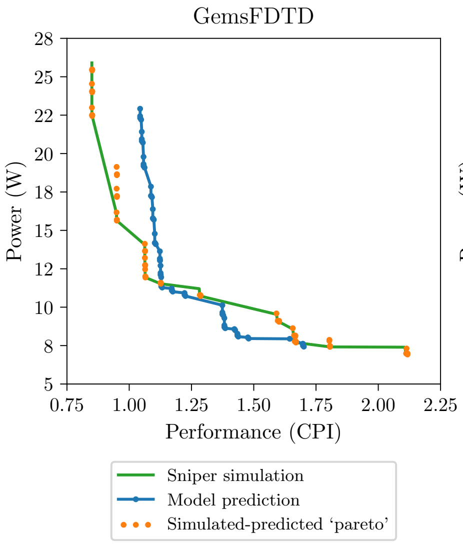
Error performance:	4.0%
Error power:	2.1%
Sensitivity:	45.2%
Specificity:	85.1%
Accuracy:	74.9%
HVR:	98.0%



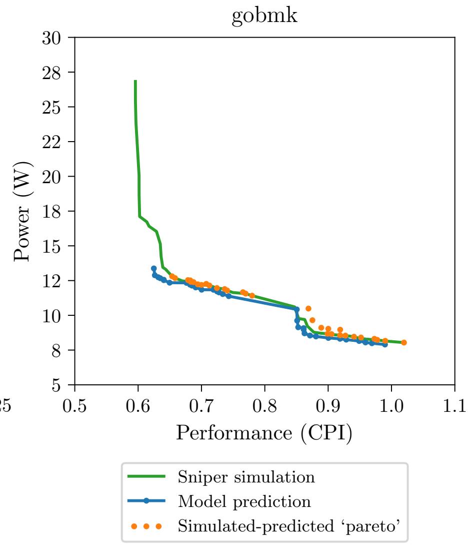
Error performance:	4.6%
Error power:	1.9%
Sensitivity:	56.7%
Specificity:	93.9%
Accuracy:	89.3%
HVR:	97.9%



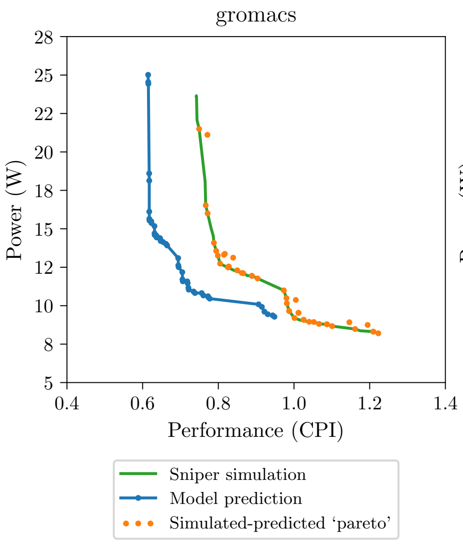
Error performance:	13.2%
Error power:	2.8%
Sensitivity:	65.4%
Specificity:	88.3%
Accuracy:	80.7%
HVR:	99.4%



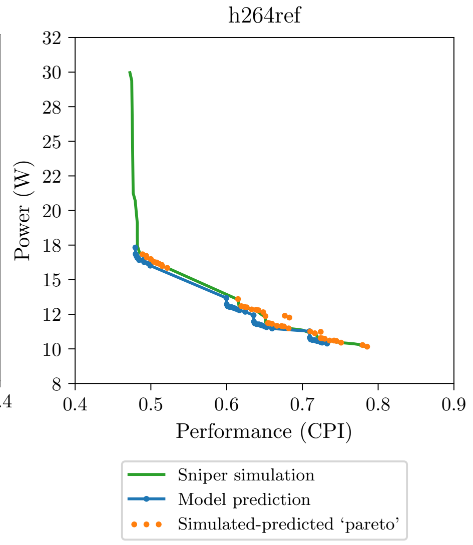
Error performance:	12.4%
Error power:	6.2%
Sensitivity:	29.0%
Specificity:	69.8%
Accuracy:	64.6%
HVR:	99.8%



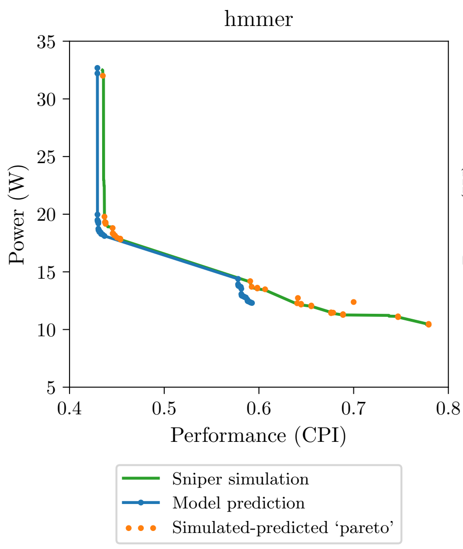
Error performance:	4.3%
Error power:	2.6%
Sensitivity:	42.2%
Specificity:	93.9%
Accuracy:	84.4%
HVR:	86.9%



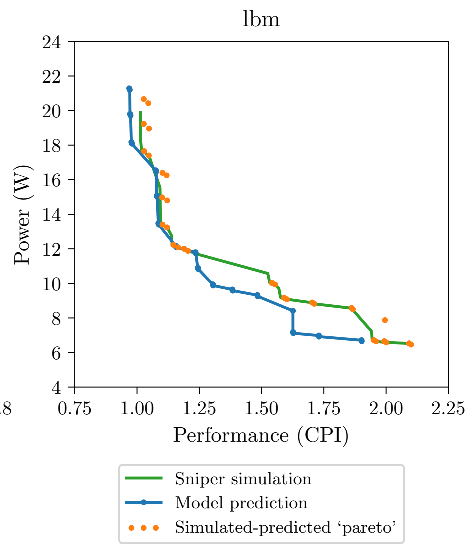
Error performance:	23.0%
Error power:	15.0%
Sensitivity:	50.0%
Specificity:	92.4%
Accuracy:	82.3%
HVR:	99.0%



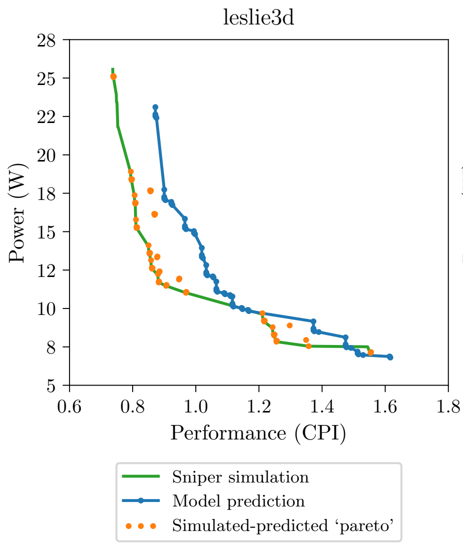
Error performance:	2.7%
Error power:	1.5%
Sensitivity:	47.5%
Specificity:	92.3%
Accuracy:	81.9%
HVR:	96.3%



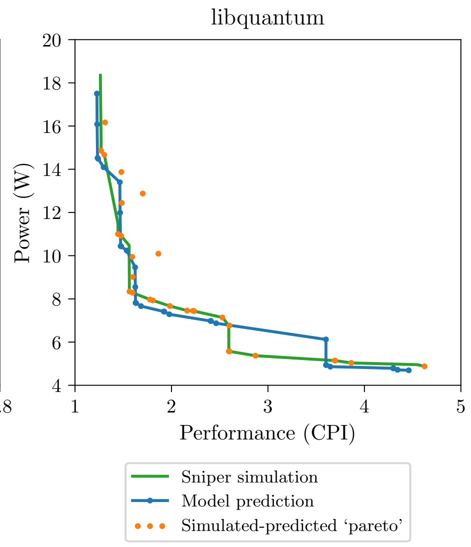
Error performance:	5.3%
Error power:	3.4%
Sensitivity:	32.2%
Specificity:	91.3%
Accuracy:	77.0%
HVR:	95.7%



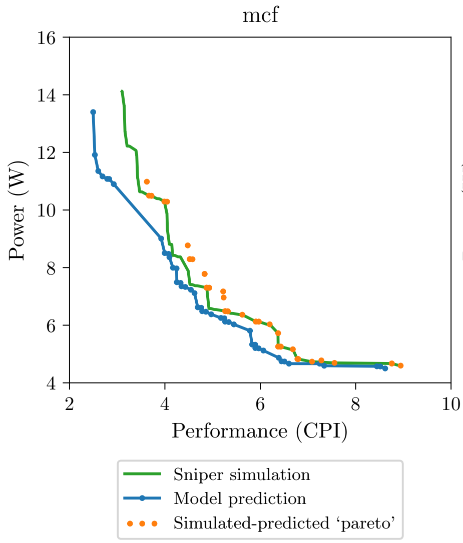
Error performance:	10.9%
Error power:	5.0%
Sensitivity:	35.4%
Specificity:	85.8%
Accuracy:	69.3%
HVR:	98.3%

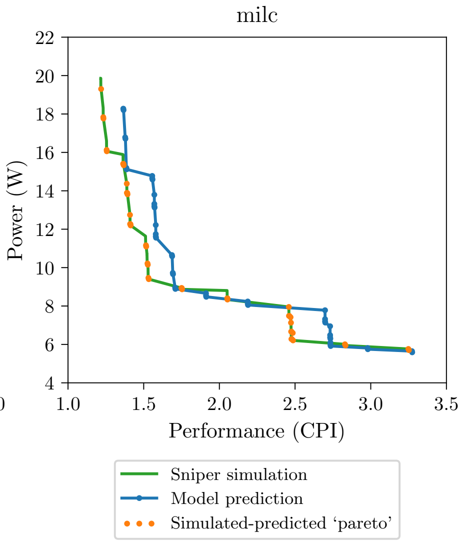


Error performance:	17.0%
Error power:	8.3%
Sensitivity:	31.8%
Specificity:	68.3%
Accuracy:	61.7%
HVR:	98.1%



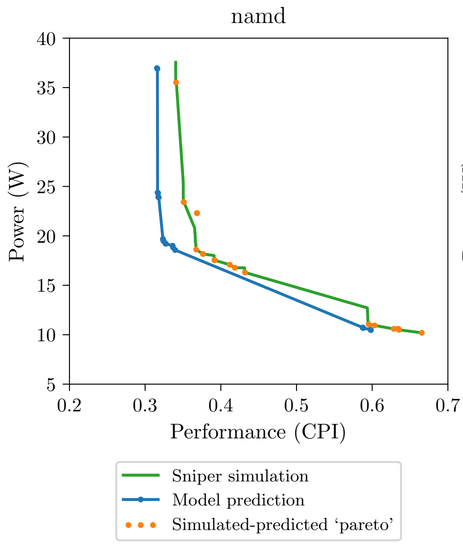
Error performance:	11.0%
Error power:	4.1%
Sensitivity:	37.0%
Specificity:	95.8%
Accuracy:	82.7%
HVR:	97.3%

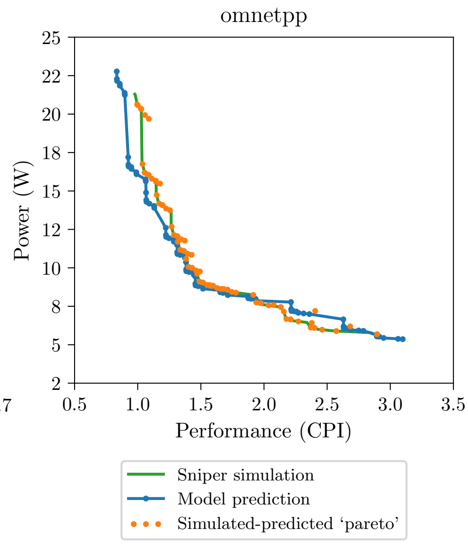




Error performance:	10.9%
Error power:	2.5%
Sensitivity:	26.3%
Specificity:	88.2%
Accuracy:	63.0%
HVR:	94.6%

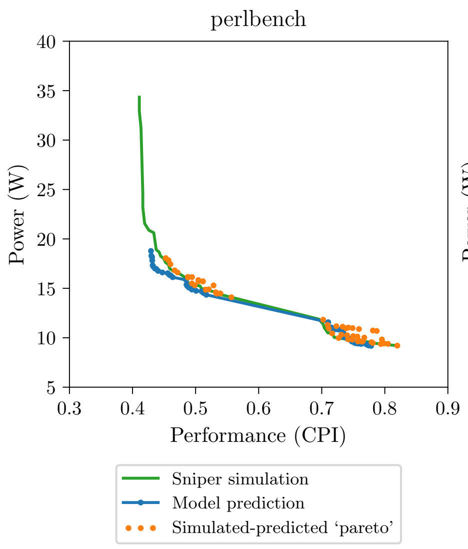
Error performance:	8.6%
Error power:	3.1%
Sensitivity:	33.7%
Specificity:	93.1%
Accuracy:	69.1%
HVR:	100.0%



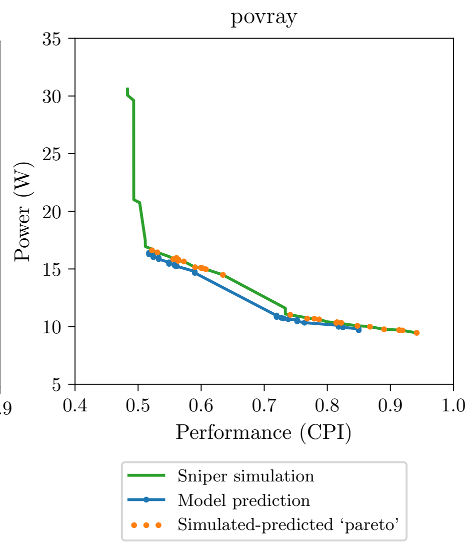


Error performance:	10.8%
Error power:	5.6%
Sensitivity:	25.0%
Specificity:	97.8%
Accuracy:	78.6%
HVR:	99.6%

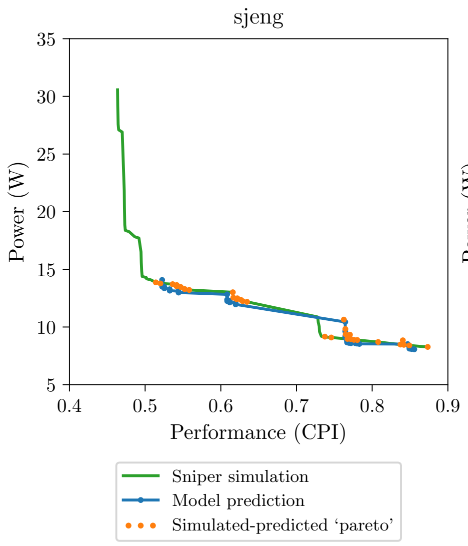
Error performance:	9.2%
Error power:	4.2%
Sensitivity:	66.3%
Specificity:	89.0%
Accuracy:	80.7%
HVR:	99.0%



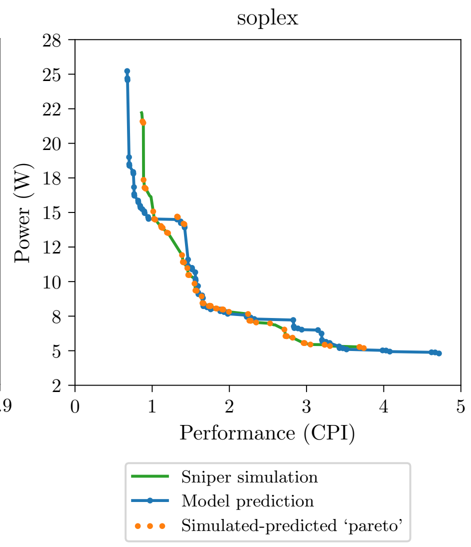
Error performance:	3.6%
Error power:	2.1%
Sensitivity:	34.8%
Specificity:	80.2%
Accuracy:	71.6%
HVR:	92.8%



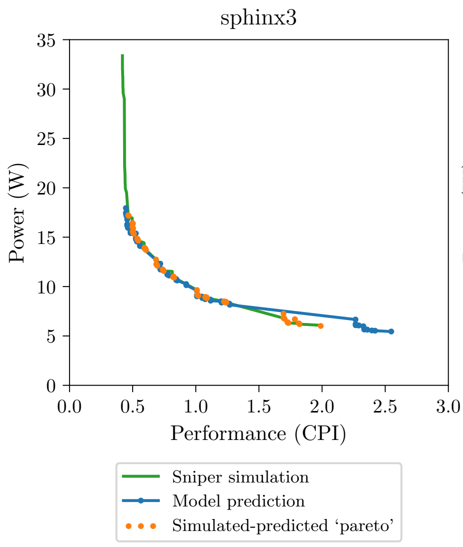
Error performance:	4.5%
Error power:	2.5%
Sensitivity:	58.3%
Specificity:	98.1%
Accuracy:	92.2%
HVR:	88.6%



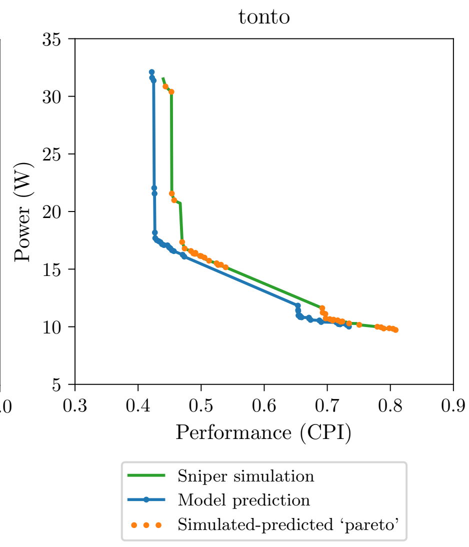
Error performance:	4.3%
Error power:	3.6%
Sensitivity:	44.2%
Specificity:	95.8%
Accuracy:	84.8%
HVR:	87.3%



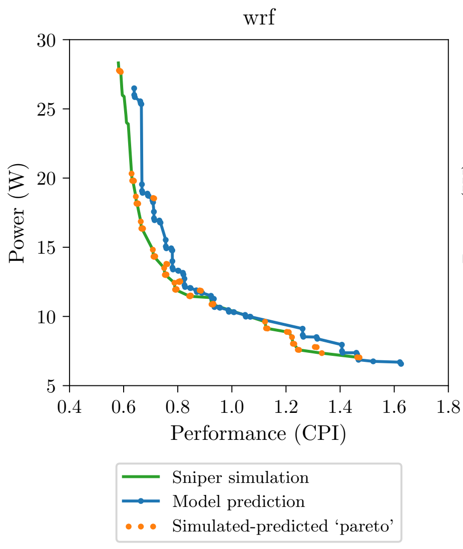
Error performance:	16.7%
Error power:	5.8%
Sensitivity:	62.1%
Specificity:	94.3%
Accuracy:	80.7%
HVR:	99.7%



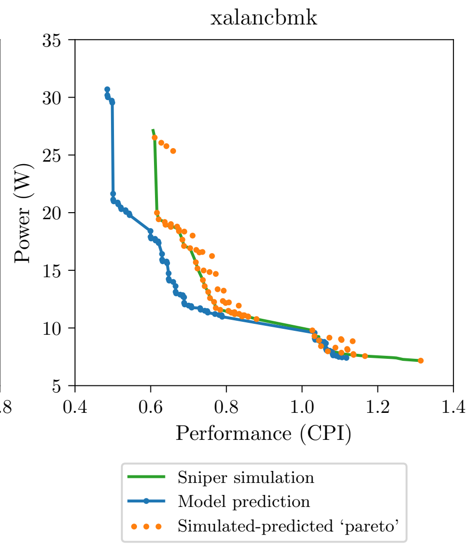
Error performance:	12.8%
Error power:	5.5%
Sensitivity:	46.3%
Specificity:	85.2%
Accuracy:	67.9%
HVR:	98.3%



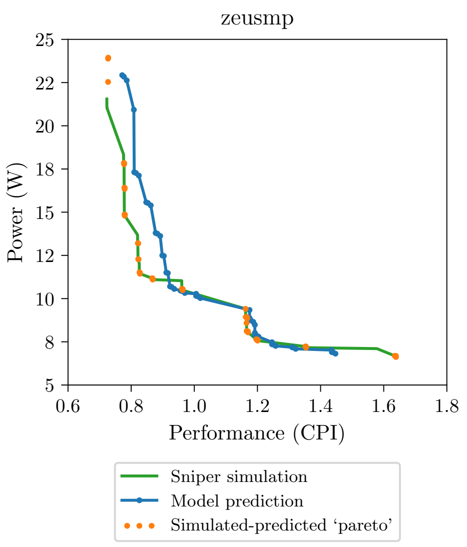
Error performance:	8.4%
Error power:	3.7%
Sensitivity:	76.3%
Specificity:	96.6%
Accuracy:	93.4%
HVR:	99.7%



Error performance:	11.7%
Error power:	6.5%
Sensitivity:	78.0%
Specificity:	83.4%
Accuracy:	82.3%
HVR:	98.5%



Error performance:	10.9%
Error power:	4.5%
Sensitivity:	70.2%
Specificity:	81.6%
Accuracy:	79.4%
HVR:	98.3%



Error performance:	7.8%
Error power:	3.9%
Sensitivity:	42.0%
Specificity:	89.7%
Accuracy:	76.1%
HVR:	99.3%

Bibliography

[1] ARM Fast Models. `https://developer.arm.com/products/system-design/fast-models/`.

[2] DynamoRIO. `http://www.dynamorio.org/`.

[3] EEMBC. `https://www.eembc.org/`.

[4] Google Protobuf. `https://developers.google.com/protocol-buffers/`.

[5] Intel X86 Encoder Decoder (XED). `https://intelxed.github.io/`.

[6] ITRS. `http://www.itrs2.net/`.

[7] LLVM. `https://llvm.org/`.

[8] Python. `https://www.python.org/`.

[9] SPEC CPU 2000. `https://www.spec.org/cpu2000/`.

[10] SPEC CPU 2006. `https://www.spec.org/cpu2006/`.

[11] O. Azizi, A. Mahesri, B. C. Lee, S. J Patel, and M. Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. In *Proceedings of the 37th annual International Symposium on Computer Architecture (ISCA)*, pages 26–36. ACM/IEEE, 2010.

[12] N. Beckmann and D. Sanchez. Modeling cache performance beyond lru. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 225–236. IEEE, 2016.

[13] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 169–180, 2005.

[14] N. Binkert, B. Beckmann, G. Black, S. K Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. volume 39, pages 1–7. ACM, 2011.

[15] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 83–94. ACM/IEEE, 2000.

[16] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. In *ACM SIGARCH computer architecture news*, volume 25, pages 13–25. ACM, 1997.

[17] T. E. Carlson, W. Heirman, S. Eyerman, Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):28:1–28:25, 2014.

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

[19] X. E. Chen and T. M. Aamodt. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. 8(3):10:1–10:28, 2011.

[20] 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. ACM/IEEE, 2004.

[21] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pages 468–477. IEEE, 1996.

[22] S. De Pestel, S. Eyerman, and L. Eeckhout. Micro-architecture independent branch behavior characterization. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 135–144. IEEE, 2015.

[23] S. De Pestel, S. Van den Steen, S. Akram, and L. Eeckhout. RPPM: Rapid performance prediction of multithreaded applications on multicore hardware. In *IEEE Computer Architecture Letters (CAL)*. IEEE, Accepted, not yet published.

[24] K. Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.

[25] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. volume 9, pages 256–268, 1974.

[26] L. Eeckhout, Y. Luo, K. De Bosschere, and L. K. John. BLRL: Accurate and efficient warmup for sampled processor simulation. volume 48, pages 451–459. Oxford University Press, 2005.

[27] D. Eklov, D. Black-Schaffer, and E. Hagersten. StatCC: a statistical cache contention model. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 551–552. IEEE, 2010.

[28] D. Eklov and E. Hagersten. StatStack: Efficient modeling of LRU caches. In *Proceedings of the International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 55–65, 2010.

[29] P. G. Emma. Understanding some simple processor-performance limits. volume 41, pages 215–232, 1997.

[30] S. Eyerman. *Analytical performance analysis and modeling of superscalar and multi-threaded processors*. PhD thesis, Ghent University, 2008.

[31] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate cpi components. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–184. ACM, 2006.

[32] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. volume 27, pages 42–53, 2009.

[33] J. W. C. Fu, J. H. Patel, and N. L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 102–110, 1992.

[34] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2010.

[35] A. Hartstein and T. R. Puzak. The optimal pipeline depth for a microprocessor. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 7–13. ACM/IEEE, 2002.

[36] J. W. Haskins and K. Skadron. Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 195–203. IEEE, 2003.

[37] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R. Stan. Hotspot: A compact thermal modeling methodology for early-stage vlsi design. In *Transactions on Very Large Scale Integration (VLSI) Systems*, volume 14, pages 501–513. IEEE, 2006.

[38] 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 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 195–206. ACM, 2006.

[39] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. Cmp$im: A pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, pages 28–36, 2008.

[40] R. Jongerius, A. Anghel, G. Dittmann, G. Mariani, E. Vermij, and H. Corporaal. Analytic multi-core processor model for fast design-space exploration. *IEEE Transactions on Computers (TC)*, 67(6):755–770, 2018.

[41] R. Jongerius, G. Mariani, A. Anghel, G. Dittmann, E. Vermij, and H. Corporaal. Analytic processor model for fast design-space exploration. In *IEEE International Conference on Computer Design (ICCD)*, pages 411–414. IEEE, 2015.

[42] 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. ACM/IEEE, 2004.

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

[44] B. C. Lee and D. M. Brooks. Illustrative design space studies with microarchitectural regression models. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 340–351. IEEE, 2007.

[45] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 249–258. ACM, 2007.

[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, 2009.

[47] D. J. Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge University Press, 2005.

[48] 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, 2005.

[49] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings of*

the International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 2–10. IEEE, 1999.

[50] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2010.

[51] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 96–105. IEEE, 2004.

[52] N. Nikoleris, A. Sandberg, E. Hagersten, and T. E. Carlson. Coolsim: Eliminating traditional cache warming with fast, virtualized profiling. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 149–150. IEEE, 2016.

[53] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–11. ACM, 1996.

[54] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 2–11, 2010.

[55] L. Peled, S. Mannor, U. Weiser, and Y. Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 285–297. ACM/IEEE, 2015.

[56] A. Sembrant, D. Black-Schaffer, and E. Hagersten. Phase guided profiling for fast cache modeling. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 175–185, 2012.

[57] Y. S. Shao and D. Brooks. Isa-independent workload characterization and its implications for specialized architectures. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 245–255. IEEE, 2013.

[58] 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. IEEE, 2001.

[59] 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. ACM, 2002.

[60] K. Singh, M. Bhadauria, and S. A. McKee. Real time power estimation and thread scheduling via performance counters. volume 37, pages 46–55. ACM, 2009.

[61] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction-window size, and cache size: Performance trade-offs and simulation techniques. In *IEEE Transactions on Computers (TC)*, volume 48, pages 1260–1281. IEEE, 1999.

[62] P. K. Szwed, D. Marques, R. M. Buels, S. A. McKee, and M. Schulz. Simsnap: Fast-forwarding via native execution and application-level checkpointing. In *Eighth Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*, pages 65–74. IEEE, 2004.

[63] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B Brockman, and N. P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 51–62. ACM/IEEE, 2008.

[64] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 392–403. ACM/IEEE, 1995.

[65] V. Uzelac and A Milenkovic. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 207–217, 2009.

[66] M. Van Biesbrouck, B. Calder, and L. Eeckhout. Efficient sampling startup for simpoint. volume 26, pages 32–42. IEEE, 2006.

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

[68] S. Van den Steen, S. De Pestel, M. Mechri, S. Eyerman, T. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. Micro-architecure independent analytical processor performance and power modeling. In *Proceedings of the International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 32–41, 2015.

[69] S. Van den Steen and L. Eeckhout. Modeling superscalar processor memory-level parallelism. *IEEE Computer Architecture Letters (CAL)*, 17(1):9–12, 2018.

[70] S. Van den Steen, S. Eyerman, S. De Pestel, M. Mechri, T. E. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. Analytical processor performance and power modeling using micro-architecture independent

characteristics. *IEEE Transactions on Computers (TC)*, 65(12):3537–3551, 2016.

[71] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, volume 33, pages 408–409. ACM, 2005.

[72] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Simulation sampling with live-points. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12. IEEE, 2006.

[73] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *International Symposium on Computer Architecture (ISCA)*, pages 84–95. ACM/IEEE, 2003.

[74] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software(ISPASS)*, pages 23–34. IEEE, 2007.