Analyse, schatting en optimalisatie van de prestatie
van computersystemen met behulp van machine learning

Analysis, Estimation and Optimization
of Computer System Performance Using Machine Learning

Kenneth Hoste

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

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen
Academiejaar 2009 - 2010

UNIVERSITEIT
GENT

ISBN 978-90-8578-373-2
NUR 980
Wettelijk depot: D/2010/10.500/49

Aan mijn vrouw Joke,
die me steeds blijven steunen is,
en onze zoon Senne,
opdat hij trots mag zijn op zijn papa.

Dankwoord

Een doctoraatsproefschrift is een werk van lange adem, dat onmogelijk tot een goed einde kan gebracht worden zonder steun van familie, vrienden en collega's. Graag bedank ik iedereen die, direct of indirect, bijgedragen heeft tot dit werk.

In het bijzonder wil ik mijn promotor prof. Lieven Eeckhout bedanken voor het ondersteunen van mijn onderzoek, de hulp bij het schrijven van artikels, het geven van goede raad en het delen van zijn professionele ervaring. Dankzij zijn begeleiding is dit werk uitgegroeid tot een volwaardig proefschrift. Daarnaast wil ik ook prof. Koen De Bosschere bedanken om mij de kans te geven om mijn doctoraatsonderzoek aan te vatten, alsook voor zijn suggesties bij het opstellen van presentaties en de manier waarop hij mededoctoraatsstudenten en andere collega's weet te motiveren.

De andere leden van mijn examencommissie wil ik eveneens uitdrukkelijk bedanken. Mijn dank gaat uit naar prof. Jan Van Campenhout, de voorzitter van de vakgroep ELIS waar ik mijn doctoraatsonderzoek heb uitgevoerd, voor zijn kritische opmerkingen; prof. Peter Dawyndt, die mijn werk vanuit een ander standpunt in detail heeft bestudeerd; en prof. Hendrik Blockeel, voor de samenwerking en zijn interessante opmerkingen en suggesties omtrent machine learning.

Furthermore, I would like to thank prof. David Kaeli to cordially accept the invitation to be part of my PhD jury despite his busy schedule, for his critical view on my work and his kind words of appreciation. Special thanks go out to dr. Diego Novillo for being part of my PhD jury, taking the time to visit Ghent and to attend the defense of this PhD dissertation, for answering my questions on GCC and related topics, and for his thoughtful questions and suggestions from an industry point of view. Thanks to both prof. Kaeli and dr. Novillo, I feel like my hard work is appreciated and recognized internationally, giving me great satisfaction and significantly boosting my self-confidence.

Prof. Erik D'Hollander wil ik bedanken om mij in te lijven als begeleider voor het vak Compilers; hierdoor heb ik ook mijn didactische vaardigheden verder kunnen ontwikkelen en werd mijn interesse voor compilers verder aangewakkerd. Bedankt aan Andy Georges voor de intense samenwerking – mede dankzij hem werden verschillende onderzoeksprojecten en artikels tot een goed einde gebracht. Op een indirecte manier was hij ook verantwoordelijk voor mijn initiële interesse in een doctoraat, waarvoor ik hem heel dankbaar ben.

Verder wil ik ook mijn andere collega's en ex-collega's bedanken voor de aangename werksfeer en de geanimeerde discussies tijdens de middagpauzes. Ik hoop dat ik jullie in de toekomst nog tegen het lijf mag lopen.

Speciale dank gaat uit naar de beheerders van BEGrid (Stijn De Smet, Bert De Vuyst en Muriel Dejonghe), en naar de beheerders van de nieuwe rekeninfrastructuur aan de Universiteit Gent (en tevens mijn toekomstige collega's) Stijn De Weirdt en Tom Kuppens. Zij stonden steeds klaar om mijn vaak lastige en dringende vragen te beantwoorden, en om oplossingen aan te bieden voor mijn noden qua rekentijd; zonder hen zou het onmogelijk geweest zijn om de resultaten die in dit proefschrift besproken worden te bekomen.

Het agentschap voor Innovatie door Wetenschap en Technologie (IWT) wil ik bedanken voor het financieren van mijn onderzoek.

Tenslote wil ik ook familie en vrienden van harte bedanken. Eerst en vooral wil ik mijn vrouw Joke bedanken voor haar geduld, steun en interesse in mijn onderzoek; ook in tijden waarin ik onterecht de illusie wekte dat mijn werk belangrijker zou zijn dan mijn gezin is ze steeds achter me blijven staan. Ook mijn 7 weken-oude zoontje Senne wil ik bedanken, voor zijn glimlach en de vreugde die hij brengt in mijn leven. Ik hoop van harte dat hij binnen enkele jaren trots zal zijn op zijn papa. Verder wil ik mijn ouders en grootouders bedanken. Zij hebben mij de kans gegeven om mijn studies aan te vangen, en zijn mij steeds met de nodige gezonde nieuwsgierigheid blijven steunen. Ook mijn schoonfamilie wil ik bedanken voor hun interesse en steun. Daarnaast bedank ik ook vrienden en kennissen die indirect hebben bijgedragen tot dit werk, alsook om mij met beide voeten op de grond te houden.

Mocht ik mensen vergeten zijn: ook jullie bedankt.

Kenneth Hoste
Gent, 30 augustus 2010

Examencommissie

Prof. Luc Taerwe, voorzitter
Decaan Faculteit Ingenieurswetenschappen
Universiteit Gent

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

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

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

Prof. Peter Dawyndt
Vakgroep TWI, Faculteit Wetenschappen
Universiteit Gent

Prof. Hendrik Blockeel
Departement Computerwetenschappen
Katholieke Universiteit Leuven

Prof. David Kaeli
Computer Architecture Research Laboratory
Northeastern University, Boston (USA)

Dr. Diego Novillo
Google Canada, Toronto (Canada)

Samenvatting

Gedurende de laatste 40 jaar heeft het vakgebied van de computerwetenschappen een verbazingwekkende vooruitgang geboekt. Terwijl de vroegste microprocessors nog ontwikkeld werden met slechts een paar duizend transistors, worden recente microprocessors geïmplementeerd met miljarden transistors. Dit heeft een exponentiële toename in de prestatie van dergelijke microprocessors teweeggebracht, door middel van verschillende innovaties in microarchitecturaal en fysisch ontwerp.

Met deze toename in prestatie nam echter ook de complexiteit van de microprocessors toe, net als de complexiteit van computersystemen in hun geheel en de computerprogramma's die er op uitgevoerd worden. Door de introductie van verschillende abstractielagen wordt ervoor gezorgd dat computeringenieurs zich kunnen concentreren op één bepaald aspect van het systeem, zonder zich bijvoorbeeld zorgen te moeten maken over hoe computerprogramma's voor dat systeem ontwikkeld zullen worden. Dit laat toe om zeer complexe doch goed presterende systemen te bouwen.

Hoewel problemen veroorzaakt door de immense complexiteit van hedendaagse computersystemen grotendeels vermeden worden door middel van abstractielagen, vormt die complexiteit nog steeds een beperkende factor in sommige opzichten: het bemoeilijkt het redeneren over de prestatie van microprocessors, het voorspellen van de prestatie ervan gegeven de karakteristieken van een bepaalde werklast en het optimaliseren van de prestatie die ze behalen voor een bepaalde groep van computerprogramma's. In dit proefschrift bestuderen we een aantal problemen omtrent de prestatie van computersystemen, waarvoor de oorzaak telkens teruggebracht kan worden tot de complexiteit van hedendaagse computersystemen. Voor elk van deze problemen stellen we een geschikte en efficiënte oplossing voor, gebruik makend van een of meerdere machine learning-technieken.

Een eerste probleem dat we bestuderen is het analyseren van het tijdsvariërende gedrag van een verzameling programma's. We tonen aan dat de hardwareprestatietellermetrieken, die sterk verbonden zijn met een bepaald hardwareplatform, niet geschikt zijn hiervoor en dat uitgemiddelde programmakarakterisatie onvoldoende detail levert. Bijgevolg stellen we een verzameling microarchitectuuronafhankelijke programmakarakteristieken voor, die toelaten om het uitvoeringsgedrag van computerprogramma's te vatten. Gebruik makende van deze karakteristieken stellen we een praktische methodologie voor om het tijdsvariërende gedrag van programma's te bestuderen. Deze methodologie maakt dankbaar gebruik van verschillende machine learning-technieken, waaronder k-means clusteren en Principale Componenten Analyse (PCA) om het meest voorkomende fasegedrag te extraheren en een genetisch algoritme om de belangrijkste programmakarakteristieken te identificeren. Door middel van kiviatdiagrammen stellen we het meest voorkomende fasegedrag visueel voor, op een compacte doch intuïtieve manier.

Hoewel vandaag het verzamelen van prestatiedata voor computersystemen relatief eenvoudig is, blijft het verwerven van inzicht vaak problematisch, of althans niet triviaal. Daarom stellen we het Processor Performance Visualizer-raamwerk voor, om prestatietrends te extraheren uit ruwe prestatiedata. Door dit te combineren met microarchitectuuronafhankelijke programmakarakteristieken, kunnen we op een snelle en eenvoudige manier de prestatietrends die in een dataset aanwezig zijn herkennen en interpreteren. We ondersteunen deze stelling door de voorgestelde methodologie toe te passen op de prestatiedata die beschikbaar is voor de SPEC CPU2000 en CPU2006 benchmark suites, wat resulteert in verscheidene interessante inzichten omtrent de relatieve prestatie van bestaande commerciële computersystemen.

Een probleem eigen aan prestatie-evaluatie en benchmarking is het anticiperen van de prestatie van computersystemen voor een bepaald programma waarin men geïnteresseerd is. Doorgaans vertrouwt men bij het rangschikken van computersystemen op de gemiddelde prestatie voor een groep van benchmarks, eventueel rekening houdende met het applicatiedomein waartoe het programma behoort en enkele hoog-niveauheuristieken, zoals of het programma rekenintensief of geheugenintensief is, het aandeel van berekeningen met gehele getallen en reële getallen, enz. Vermits deze manier van werken vaak aanleiding geeft tot onnauwkeurige prestatieschattingen, stellen we een methodologie voor die prestatieschatting mogelijk maakt op basis van mi-

croarchitectuuronafhankelijke programmakarakteristieken en een combinatie van verschillende machine learning-technieken, in het bijzonder een genetisch algoritme en k-nearest-neighbors. Na het quantificeren van de relevantie van elke programmakarakteristiek met betrekking tot de prestatieverschillen voor een groep van benchmarks, krijgt elke programmakarakteristiek op een overeenkomstige manier een gewicht toegekend. Om dan de prestatie voor een bepaald computerprogramma te schatten, wordt een gewogen gemiddelde berekend van de prestatiemetrieken van de benchmarks met het meest gelijkaardige programmagedrag. Door middel van de SPEC CPU benchmarkprogramma's en een kruisvalidatie-evaluatie, tonen we aan dat ons raamwerk tot betere rangschikkingen van computersystemen leidt in vergelijking met die op basis van gemiddelde prestatie.

Naast het analyseren en voorspellen van de prestatie van computersystemen, gaan we ook na hoe het specialiseren van systeemsoftware op een automatische manier kan gebeuren. Statische compilers bieden meestal een aantal optimalisatieniveaus aan die elk op zich een afweging maken tussen een aantal metrieken, zoals compilatietijd, codegrootte en codekwaliteit. Heden ten dage zoeken compilerontwikkelaars hun toevlucht tot hun ervaring en eenvoudige heuristieken bij het samenstellen van deze optimalisatieniveaus, wat resulteert in een tijdrovend en arbeidsintensief proces. Bovendien vereist dit proces diepgaande kennis over het grote aantal mogelijk op elkaar inwerkende compileroptimalisaties. Om dit probleem aan te pakken stellen we COLE voor, een raamwerk dat vertrouwt op een evolutionair zoekalgoritme om automatisch compileroptimalisatieniveaus samen te stellen, voor een bepaalde verzameling van computerprogramma's en een bepaald hardwareplatform. Door middel van de GNU Compiler Collection (GCC) software en de SPEC CPU benchmarks tonen we aan dat het raamwerk veel beter presteert dan willekeurig zoeken en dat de verkregen optimalisatieniveaus betere afwegingen voorstellen dan handmatig samengestelde niveaus. We stellen experimentele resultaten voor die aantonen dat het specialiseren van bepaalde optimalisatieniveaus voor een bepaald hardwareplatform en een bepaalde verzameling van programma's belangrijk is om goede afwegingen te verkrijgen. Daarnaast analyseren we de samenstelling van de niveaus om interessante inzichten te verkrijgen omtrent het nut van individuele compileroptimalisaties.

We gaan ook na hoe een Just-In-Time-compiler of JIT-compiler gespecialiseerd kan worden voor een bepaald hardwareplatform en een

bepaalde verzameling van computerprogramma's. Deze taak vormt een grotere uitdaging dan het samenstellen van individuele optimalisatieniveaus voor een statische compiler. Een moderne JIT-compiler maakt immers gebruik van meerdere optimalisatieniveaus om een dynamisch optimalisatiemechanisme te implementeren. Dit gebeurt in combinatie met een dynamische regelaar, die beslist welke delen van de programmacode geoptimaliseerd worden met behulp van welk optimalisatieniveau en ook wanneer de optimalisatie wordt uitgevoerd. Dit proces brengt enkele subtiele afwegingen met zich mee en creëert complexe interacties tussen de verschillende optimalisatieniveaus en de parameters die de dynamische regelaar aansturen. Steunend op het COLE raamwerk voor statische compilers, stellen we een volledig geautomatiseerd afstelraamwerk voor JIT-compilers voor. De methodologie bestaat uit twee stappen. Eerst wordt een selectie van optimalisatieplannen gemaakt, waarna verschillende JIT-compilers die een aantal van die plannen gebruiken, worden samengesteld, geëvalueerd en afgesteld. Door experimentele evaluatie, gebruik makend van de Jikes RVM software en een verzameling Java benchmarks, tonen we aan dat het raamwerk in staat is JIT-compilers te configureren die gemiddeld gezien even goed presteren als een manueel afgestelde JIT-compiler. Het raamwerk laat ook toe om een JIT-compiler te specialiseren voor één bepaald computerprogramma, wat tot significante versnellingen leidt in vergelijking met een JIT-compiler die afgesteld werd voor gemiddelde prestatie.

Summary

In the last 40 years, the field of computer engineering has shown amazing progress. While the early microprocessors were built using only a few thousand transistors, recent microprocessors are implemented using billions of transistors. This has led to an exponential increase in the performance delivered by these microprocessors, through various innovations in both microarchitectural and physical design.

Along with these performance improvements however, the complexity of microprocessors also increased, just like the complexity of computer systems overall and the software applications that run on them. By introducing different layers of abstraction, computer engineers are able to focus on one particular aspect of the system, without having to worry about how applications for that system will be developed for example. This allows for building complex but well performing systems.

Although problems caused by the high complexity of modern computer systems are mostly avoided by the use of layers of abstraction, complexity is still a limiting factor in some respects: it makes it hard to reason about the performance of modern microprocessors, anticipate how they will perform given the characteristics of a certain workload of interest, and optimize the performance they achieve for a particular (set of) workload(s). In this dissertation, we study a number of problems related to computer system performance for which the root cause can be brought back to the complexity of modern systems. For each problem, we present an adequate solution using one or multiple machine learning techniques.

A first problem we look into is analyzing the time-varying behavior of a set of applications. We show that using hardware performance counter metrics that are specific to a particular hardware platform are ill-suited for this, and that aggregate workload characterization pro-

vides insufficient detail. Therefore, we present a set of microarchitecture-independent workload characteristics, which allow for capturing the program behavior of applications. Using these characteristics, we propose a feasible methodology for studying the time-varying program behavior. This methodology employs multiple machine learning techniques, including k-means clustering and Principal Component Analysis (PCA) to determine the most prominent phase behaviors, and genetic algorithms to identify the key workload characteristics. Using kiviat diagrams, we represent these prominent phase behaviors visually on a concise though intuitive manner.

While collecting performance numbers for computer systems is relatively simple at present, obtaining insight into the data remains troublesome, or at least non-trivial. Therefore, we present the Processor Performance Visualizer framework, which relies on Principal Component Analysis to extract performance trends from a set of raw performance numbers. By combining this with microarchitecture-independent workload characteristics, we are able to quickly and easily recognize and interpret the performance trends captured by the data set. We support this claim by applying the methodology to the performance data available for the SPEC CPU2000 and CPU2006 general-purpose benchmark suites, resulting in various interesting insights regarding the relative performance of existing high-end commercial computer systems.

A ubiquitous problem in performance evaluation and benchmarking is anticipating the performance of computer systems for a particular application-of-interest. Current practice mainly relies on ranking computer systems based on the average performance across a set of benchmarks, at best incorporating the application domain of the application-of-interest and/or some high-level heuristics, e.g., whether the application is compute-intensive or memory-intensive, the ratio of integer and floating-point operations, etc. Because this may lead to inaccurate performance estimations, we present a performance estimation methodology based on microarchitecture-independent workload characteristics and a combination of machine learning techniques, more particularly genetic algorithms and k-nearest-neighbors. After quantifying the relevance of each workload characteristic with respect to the performance differences observed for a set of benchmarks, we weight each workload characteristic accordingly. To estimate the performance for the application-of-interest, we compute the weighed average of the performance numbers of the similarly behaving benchmarks. Using the

SPEC CPU benchmarks in a cross-validation setup, we show that our framework yields significantly better rankings of computer systems as opposed to current practice which is based on average system performance.

Next to analyzing and anticipating computer system performance, we also look into automatically tuning system software. Static compilers usually provide a number of optimization levels representing different trade-offs between a number of objectives like compilation cost, code size and code quality. Currently, compiler developers resort to experience and high-level heuristics when constructing these optimization levels, resulting in a time-consuming and labor-intensive process. It also requires in-depth knowledge about the multitude of potentially interacting compiler optimizations. To alleviate this issue we present COLE, a framework that relies on a multi-objective evolutionary search algorithm to automatically construct compiler optimization levels for a particular set of applications and a particular hardware platform. Using the GNU Compiler Collection (GCC) and the SPEC CPU benchmarks, we show that our framework significantly outperforms random searching and the obtained optimization levels represent better trade-offs compared to manually constructed levels. We present empirical evidence that specializing optimization levels for a particular hardware platform and set of applications is important for obtaining good trade-offs, and we analyze the composition of the obtained levels which reveals interesting insights concerning the usefulness of individual compiler optimizations.

We also look into tuning a Just-In-Time (JIT) compiler to a particular hardware platform and set of applications. This is a more challenging task than constructing optimization levels for a static compiler. Modern JIT compilers use multiple optimization levels to implement a dynamic optimization mechanism. This is combined with an adaptive controller, which decides what parts of the application code are optimized using which optimization level, and when the optimization is performed. This process involves subtle trade-offs, and creates complex interactions between the various optimization levels and the parameters that steer the adaptive controller. Building on the COLE framework for static compilers, we propose a fully automated tuning framework for JIT compilers. The methodology uses a two-step approach. A set of suitable optimization plans is obtained, after which JIT compilers using a number of these plans are constructed, evaluated and tuned. Through experimental evaluation using the Jikes RVM and a collection of Java

benchmarks, we show that the framework is able to deliver JIT compilers that perform as good as a manually tuned JIT compiler on average. The framework also allows for tuning a JIT compiler to a particular application, which is shown to yield significant speedups compared to a JIT compiler tuned for average performance.

Contents

List of Tables

List of Figures

List of Abbreviations

AOS	Adaptive Optimization System
API	Application Programming Interface
CMP	Chip Multi-Processor
CPI	Cycles-Per-Instruction
CPU	Central Processing Unit
GA	Genetic Algorithm
GAg	global branch history, global prediction table
GAs	global branch history, local prediction table
GC	garbage collector
GCC	GNU Compiler Collection
HPC	Hardware Performance Counter
ILP	Instruction-Level Parallelism
IPC	Instructions-Per-Cycle
ISA	Instruction Set Architecture
JIT	Just-In-Time
kNN	k-nearest-neighbors
LRU	Least-Recently Used
MICA	Microarchitecture Independent Characterization of Applications
PAg	local branch history, global prediction table
PAs	local branch history, local prediction table
PCA	Principal Component Analysis
PPV	Processor Performance Visualizer
PPM	Prediction by Partial Match
TLB	Translation Lookaside Buffer

Chapter 1

Introduction

Any sufficiently advanced technology is indistinguishable from magic.
Arthur C. Clarke

In the last 40 years, the field of computer engineering and more specifically microprocessor design has shown amazing progress. The Intel 4004 microprocessor which was introduced in 1971 was implemented using 2,300 transistors [61]. Recent microprocessors are built using on the order of a few billion transistors, an increase of over six orders of magnitude, in line with Moore's law [60]. Examples of modern microprocessors include the Intel Core i7 implementing the Nehalem microarchitecture (781 million transistors) and the Intel Itanium 9300 (2 billion transistors).

The exponential growth in the number of transistors was accompanied by an exponential increase in the performance delivered by these microprocessors. An indicative measure is the clock frequency: while the Intel 4004 ran at 750 kHz, modern microprocessors run at frequencies of up to 5 GHz[1]. Various important innovations in microarchitectural design contributed to the impressive performance gains alongside advances in chip technology: pipelining, superscalar execution, out-of-order execution, branch prediction and speculative execution, caches, etc. In recent years, the ever growing transistor budget was spent implementing chip-multiprocessors (CMPs, also called multi-core processors), because of diminishing returns in terms of performance and huge power costs of innovations targeted at further improving single-thread performance.

[1]IBM POWER6, see `http://www.ibm.com/systems/power/hardware/595/specs.html`.

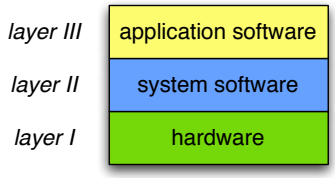


Figure 1.1: The three main layers of abstraction: hardware, system software and application software.

The key contributor to this success story is the use of various *layers of abstraction*. By introducing layers of abstraction, both hardware manufacturers and software developers have been able to work in their own field of expertise more or less independently, allowing for great improvements in overall computer system performance and providing support for previously inconceivable software applications on these systems.

For example, a computer engineer only needs a high-level view of the types of applications that will run on the system he/she is designing. By specifying an Instruction Set Architecture (ISA), which serves as a layer of abstraction between the hardware and the end user's software applications, he/she leaves it up to the system software developers to provide support for software that is going to run on the system being designed. Likewise, a software developer only needs a high-level understanding of how a microprocessor is able to execute billions of instructions per second in order to design an application for it.

Figure 1.1 shows the three main layers of abstraction in computer systems:

(I) the hardware layer, consisting of the physical microprocessor implemented using semiconductor technology and its peripherals (e.g., main memory, I/O, etc.);

(II) the system software layer, consisting of the operating system (OS), compilers for different programming languages, runtime systems (e.g., a Java Virtual Machine) and middleware, etc.;

(III) the application software layer, consisting of the end-user software applications.

Each of the layers can be further divided, but this limited set of layers is sufficient for the discussion to follow.

Although abstraction has helped in achieving great performance in modern computer systems, it does not take away their enormous complexity. Modern microprocessors are utterly complex pieces of technology by themselves, with a large number of structures each responsible for a piece of the puzzle. These structures potentially have a mutually large impact on efficiency, e.g., the cache configuration may have a significant impact on the effectiveness of the processor pipeline width and depth. This makes it very hard to reason about the performance of modern microprocessors, analyze their performance under a given workload, anticipate how they will perform giving the characteristics of a workload of interest, or optimize the performance achieved for a particular (set of) workload(s).

In this dissertation, we look at several problems for which the root cause can be attributed to the sheer complexity of modern computer systems, and more specifically the microprocessors that are at the heart of these systems and the software that runs on them. In the solutions presented for each of the problems addressed, we rely on a variety of machine learning techniques to tackle the problems efficiently and adequately. Combining several well-chosen machine learning techniques, each of which taking care of a particular aspect of the problem, is shown to yield an appropriate solution for the problem at hand.

Before describing the contributions made in this dissertation, we first briefly revisit machine learning.

1.1 Machine learning

Machine learning is a field of computer science that focuses on designing algorithms and techniques that automatically 'learn' about a particular problem. The term learning is interpreted broadly in this context: it may refer to discovering structure in a data set, or to training a so-called model that is able to estimate some quantitative measure of previously unobserved instances.

In general, machine learning techniques are split into two categories: supervised techniques and unsupervised techniques. Supervised machine learning techniques aim at producing some kind of model based on training data, consisting of a set of training instances described by feature vectors and their corresponding output values. Usually, the end goal is to obtain a model that is able to accurately estimate the output values for unseen instances, i.e., that generalizes well

beyond the provided training data. Examples of supervised techniques include decision trees and the k-nearest neighbor algorithm. Unsupervised machine learning techniques on the other hand operate on unlabeled data points. These techniques aim at deducing information about the structure of the data set provided, and often rely heavily on preprocessing. Examples include clustering (e.g., k-means clustering) and Principal Component Analysis (PCA).

It should be noted that there is no strict set of rules for determining whether or not a particular technique is a machine learning technique. In this dissertation, the term machine learning is used in a broad sense. We consider evolutionary algorithms and more in particular genetic algorithms to be machine learning techniques too, while these are actually search or optimization algorithms. Likewise, PCA is often viewed as a data mining or data analysis technique rather than a machine learning technique.

For an in-depth overview of the field of machine learning, we refer to [4, 78].

1.2 Contributions

This thesis makes a number of contributions. In each of the following chapters, we discuss a specific problem and the solution we came up with. The different contributions are situated in one of the abstraction layers shown in Figure 1.1. Chapter 2 is situated in the application software layer, Chapters 3 and 4 look into problems related to the hardware abstraction layer, and Chapters 5 and 6 concern problems in the system software layer.

1.2.1 Analyzing and estimating performance

Although collecting performance numbers for computer systems and data describing the low-level behavior of applications is relatively simple, gaining insight and extracting relevant information is non-trivial. Likewise, anticipating the performance of a particular system for a given application-of-interest is hard. Introducing multiple layers of abstraction has made designing these systems and developing software for them feasible, but the root cause behind the problems mentioned, i.e., the complexity of modern computer systems and the software that runs on them remains. In the first three chapters following this intro-

duction, we look into problems related to analyzing and estimating computer system performance.

Analyzing time-varying program behavior

While software developers mostly care about the features of the software applications they build and computer engineers are focused on spending the transistor budget adequately and efficiently, both are concerned about the performance of the computer system under consideration. The key to recognizing and resolving performance bottlenecks lies in evaluating the performance of the computer system using the set of applications-of-interest. Understanding the characteristics of the various workloads that will run on the system is important; that way, a computer engineer can reconsider certain design decisions, while a software developer can realize that some part of a software application should be rewritten in order to make it a better fit for a particular target system.

Although multiple benchmark suites each representing a certain application domain are available, it is still hard to obtain general insight in the different inherent program behaviors for a particular set of applications. Current workload characterization studies tend to focus on using performance metrics that are tied to the computer system under study. Also, most studies tend to limit themselves to looking at the average program behavior. In Chapter 2, we show that both these common practices potentially yield misleading results. We present a set of microarchitecture-independent workload characteristics that allow to capture the true inherent program behavior.

Subsequently, we present a methodology for studying the time-varying program behavior of a set of applications. This methodology relies on several machine learning techniques including clustering, Principal Component Analysis and genetic algorithms. The methodology is applied to a data set of workload characteristics for five different benchmark suites. Analyzing the results leads to various interesting insights on the coverage, uniqueness and diversity of existing benchmark suites.

This workload characterization methodology was published in:

- Kenneth Hoste and Lieven Eeckhout, "Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics", in

Proceedings of the 2006 IEEE International Symposium on Workload Characterization (IISWC). IEEE Computer Society, 2006, pp. 83–92.

- Kenneth Hoste and Lieven Eeckhout, "Microarchitecture-Independent Workload Characterization", in *IEEE Micro, Special Issue on Hot Tutorials, IEEE Computer Society, Vol. 27 (3), 2007, pp. 63–72.*

- Kenneth Hoste and Lieven Eeckhout, "Characterizing the Unique and Diverse Behaviors in Existing and Emerging General-Purpose and Domain-Specific Benchmark Suites", in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE Computer Society, 2008, pp. 157–168.*

Recognizing and interpreting performance trends

For several benchmark suites, performance data is publicly available for a wide range of computer systems. Examples are the performance databases for commercial machines that have been constructed by the SPEC consortium. A well-known example is the set of performance numbers available for the SPEC CPU benchmark suites; for both the SPEC CPU2000 and CPU2006 benchmark suites, performance data is available for over 1,000 systems, providing a wealth of information. However, quickly gaining insight into how these systems differ from each other and which performance trends can be observed is difficult.

To alleviate this problem, we present the Processor Performance Visualizer methodology in Chapter 3. In this work, we combine Principal Component Analysis, which automatically extracts performance trends, with the earlier proposed microarchitecture-independent workload characteristics, which allow for giving meaningful interpretations to the observed trends. We apply the proposed methodology to the performance databases for both the SPEC CPU2000 and CPU2006 benchmark suites. We observe and analyze various interesting performance trends, which both confirm a number of well-known trends and bring forth new insights.

This work was published in:

- Kenneth Hoste and Lieven Eeckhout, "A Methodology for Analyzing Commercial Processor Performance Numbers", in *IEEE Computer, Vol. 42 (10), 2009, pp. 70–76.*

Estimating relative computer system performance

Anticipating the performance of a computer system is infeasible without thorough performance evaluation of the system, given the complexity we already discussed. Evaluating systems using a set of benchmarks yields valuable information on the performance of these systems relative to each other.

However, this usually does not allow for ranking these systems in terms of performance for a particular application-of-interest. Only in rare cases is the application-of-interest part of the benchmark suite, and whenever this is not the case, the user is forced to resort to the average performance across the benchmark set. At best, a rough performance estimation can be made by using high-level heuristics based on coarse program characteristics, e.g., whether the application is memory-intensive or compute-intensive, and which benchmarks are similar in that respect.

In Chapter 4, we propose a performance estimation methodology based on microarchitecture-independent workload characteristics and a combination of machine learning techniques such as genetic algorithms and k-nearest-neighbors. First, we weigh each of the workload characteristics based on their relevance with respect to performance differences. Subsequently, we are able to identify the benchmarks which are most relevant for our application-of-interest based on the weighted workload characteristics. This way, we obtain a more accurate estimate of the relative performance differences between the computer systems under consideration. We evaluate the methodology using the SPEC CPU2000 and CPU2006 benchmarks, and show that significant improvements in the ranking of the systems in terms of the performance for the application-of-interest are obtained, compared to current practice which is based on average system performance.

This methodology was published in:

- Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K. John and Koen De Bosschere, "Performance Prediction based on Inherent Program Similarity", in *Proceedings of the 15th international conference on Parallel Architectures and Compilation Techniques (PACT), ACM. 2006, pp. 114–122.*

1.2.2 Automatically specializing system software

Apart from the relationship between the inherent behavior of software applications and the hardware of the computer system it is running on, also the system software that forms the link between both layers is affected by the complexity of modern computer systems.

Typically, system software tools, e.g., compilers and runtime systems such as a Java Virtual Machine (JVM), go further than merely making sure that the application software runs (correctly) on the underlying hardware. Most system software also fulfills the task of trying to optimize the applications, which are mostly oblivious to idiosyncrasies of the hardware. Unfortunately, achieving optimal performance is a challenging task. Compiler optimization is sometimes referred to as a "black art", precisely because of the difficulty in optimizing a particular application for a particular hardware platform.

In Chapters 5 and 6, we present two frameworks for performing tedious but critical tasks for system software in order to be able to deliver good performance.

Constructing optimization levels for a static compiler

Static compilers usually provide a number of optimization levels (e.g., `-O1`, `-O2`, `-O3`, `-Os`), each a combination of optimizations representing a trade-off between different objectives, e.g., compilation time, code quality and code size. These standard optimization levels allow software developers and end-users to produce heavily optimized builds of their applications, without having to select individual optimizations.

Defining these standard optimization levels is a challenging task however, for a variety of reasons. Most compilers offer a large number of optimizations, resulting in a huge amount of possible optimization levels; e.g., 30 optimizations already result in over one billion possible optimization levels. Also, optimizations potentially affect the applicability and/or efficacy of other optimizations, and different objectives may be affected conflictingly by a particular optimization. On top of this, the effect of a compiler optimization is highly dependent on the code being compiled and the hardware platform for which the code is being compiled.

In current practice, compiler developers construct a number of compiler optimization levels manually, relying heavily on experience, intu-

ition and high-level heuristics. This is a tedious and labor-intensive task, which requires deep knowledge of the various optimizations and their possible interactions. It also requires taking several conservative decisions, to ensure that the optimization level performs well for a variety of applications and hardware platforms.

To address this issue we present COLE in Chapter 5, a framework based on an evolutionary search algorithm that allows for automatically constructing a set of optimization levels representing trade-offs between a number of objective functions. Using the GNU Compiler Collection (GCC), we show that the optimization levels obtained using our fully automated framework significantly outperform the manually constructed standard optimization levels, and that the evolutionary search algorithm clearly outperforms random searching. Studying the optimization levels obtained through our framework in terms of their constituent optimizations yields various interesting insights.

This work was published in:

- Kenneth Hoste and Lieven Eeckhout, "COLE: Compiler Optimization Level Exploration", in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO), ACM, 2008, pp. 165–174.*

Automatically tuning a Just-In-Time compiler

In recent years, managed programming languages such as Java have gained a lot of interest, because of the cross-platform portability they offer to application software developers. This portability is achieved by compiling the applications to an intermediate machine-independent level, called bytecode, and providing a process virtual machine (e.g., a Java Virtual Machine (JVM)) that executes the application from this bytecode. Besides portability, this approach has an additional advantage: it allows for the use of runtime information to really focus the optimization of the application at runtime. Typically, a JVM will recompile frequently executed code using a more aggressive optimization strategy at runtime, a mechanism known as Just-In-Time (JIT) compilation, thereby gradually improving the performance during the execution of the application.

Modern JIT compilers use multiple optimization levels to implement this dynamic optimization mechanism, in combination with an adaptive controller that decides which parts of the application are

recompiled to which level, and when the recompilation is to be performed. In this process, trade-offs need to be made between the cost of recompilation and the expected benefit in terms of performance, since the recompilation adds extra overhead at runtime.

Tuning a dynamic JIT compiler for optimal performance is even more challenging than coming up with individual optimization levels for a static compiler. Besides the difficulties in constructing suitable optimization levels that deliver useful trade-offs as discussed before, the inherent complexity of the optimization mechanism of a JIT compiler complicates things further. In a JIT compiler, the different optimization levels potentially have a significant effect on each other. Indeed, the decision whether or not to recompile a piece of code is influenced by the expected speedup, which in turn depends on the quality of the currently used binary version of that piece of code and the additional speedup delivered by the higher optimization levels. In addition, the adaptive controller also has some knobs to turn, for example to adjust the notion of frequently executed code.

Building on the COLE framework, we present a fully automated tuning framework for JIT compilers in Chapter 6. The methodology uses a two-step approach to deal with the complexity of the optimization strategy of a JIT compiler. First, a set of suitable optimization levels is determined. The second step uses these optimization levels to construct, evaluate and tune JIT compilers that perform well for one or multiple applications on the target platform. Experimental evaluation using a set of Java benchmarks and the Jikes RVM shows that the framework is able to deliver JIT compilers that achieve performance that is competitive to a manually tuned JIT compiler. Using the automatic tuning framework, it becomes straightforward to specialize a JIT compiler to a single application-of-interest or a target hardware platform, which is shown to yield significant speedups over a manually tuned default JIT compiler.

This work was published in:

- Kenneth Hoste and Lieven Eeckhout, "Automated Just-In-Time Compiler Tuning", in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO), ACM, 2010, pp. 62–72.*

Chapter 2

Phase-level Microarchitecture-Independent Workload Characterization

You can have data without information,
but you cannot have information without data.
Daniel Keys Moran

Computer workloads are ever evolving. Software companies and researchers continuously come up with new applications, often from new application domains, e.g., multimedia, gaming, data mining, physics simulation, etc. Not only are these emerging workloads induced by customer demands or innovation in software, but also by advances in technology which result in increased computing power. This opens up opportunities for workloads that were never considered before.

Consequently, it is important for a computer system designer to understand the characteristics of these (emerging) workloads, so that the computer system under design is optimized for its target workloads. Designing a future computer system using yesterday's applications may lead to a suboptimal design for its future workloads [109].

Computer architects and performance analysts are well aware of this and are therefore continuously looking for new emerging workloads. Whenever an emerging workload is identified, a number of *benchmarks* are collected that represent this emerging workload. Benchmarks are no different from real applications, except that significant efforts are made to ensure that the applications are portable and ro-

bust, to include multiple input data sets of different sizes, and to provide an evaluation framework. This way, performance numbers for collections of these benchmarks – called *benchmark suites* – can be used for evaluating new microprocessor designs and comparing these with previous designs. Well-established benchmark suites in the field of computer architecture and compilers include SPEC CPU (general-purpose computing) [51, 87] and EEMBC (embedded systems) [88]. Examples of recently introduced benchmark suites covering emerging workloads are BioPerf (bioinformatics) [12], BioMetricsWorkload (biometrics) [25], BioInfoMark (bioinformatics) [71], MineBench (data mining) [80], PhysicsBench (physics simulation) [105], ImplantBench (bio-implantable computing) [63], MediaBench II (multimedia) [41], DaCapo (Java client) [16], STAMP (transactional memory) [77] and PARSEC (recognition, mining and synthesis (RMS)) [14].

In an early design stage, computer architects rely heavily on software simulation of microprocessor designs, because building hardware prototypes for each possible design point is simply too expensive. A big disadvantage of simulation is that it is several orders of magnitude slower than native hardware execution [24]. This implies that including new benchmark suites in the design cycle has to be considered carefully. If the runtime behavior of a new benchmark suite is not significantly different from the runtime behavior of benchmark suites already included in the design process, then there is no need for including these new benchmarks. Simulating those additional benchmarks would only add to the overall simulation time, without providing additional insight about the performance of the design.

Therefore, it is important to assess how different the workload characteristics of these new workloads are compared to those of already existing benchmark suites. This will provide insight into whether the next-generation microprocessors need to be designed differently compared to today's machines. It will also be very helpful in managing the simulation cost, without compromising the scope and accuracy of the performance evaluation of the microprocessor design. Moreover, the need for a solid *workload characterization methodology* is increasingly important given the current shift to chip-multiprocessor (CMP) computing — especially for heterogeneous CMPs with different cores being specialized to particular types of workloads.

As we will show in this chapter, workload characterization is not a straightforward task. A traditional workload characterization

methodology may produce misleading results because they depend on workload characteristics that are specific to the particular hardware on which they are obtained. Also, many workload characterization studies limit themselves to aggregate workload characterization which fails to capture the time-varying behavior of workloads. The more informative phase-level workload characterization yields large amounts of data, which makes it more difficult to obtain insight.

We address these problems by presenting a new phase-level workload characterization methodology. We present a set of microarchitecture-independent workload characteristics in Section 2.1, to solve the problems with the traditional hardware performance counter based workload characteristics. In Section 2.2, we outline a feasible way of using these workload characteristics to analyze the time-varying phase-level runtime behavior of workloads. We apply this methodology to a set of five benchmark suites in Section 2.3, allowing for a detailed comparative study of the diversity and uniqueness of the various benchmarks.

In this work, powerful machine learning techniques such a principal component analysis, genetic algorithms and cluster analysis will prove to be very useful in gaining insight into inherent workload behavior.

2.1 Microarchitecture-independent workload characterization

In the following sections, we will describe the traditional workload characterization methodology which relies on hardware performance counters (Section 2.1.1), discuss the pitfall in using these workload characteristics to compare inherent program behavior (Section 2.1.2) and present our set of microarchitecture-independent workload characteristics as an alternative (Section 2.1.3).

2.1.1 Hardware performance counter based workload characterization

Most modern microprocessors include special purpose registers and accompanying hardware logic called *hardware performance counters* (HPCs) [96]. These allow for low-level performance analysis, by keep-

ing track of the frequency count in which certain performance-related events occur during the execution of a program [10]. Because the counters are built into hardware, the overhead for collecting performance metrics using HPCs is negligible, which makes them very attractive for performance analysis on real hardware.

Studies on benchmark suites representing emerging workloads often use these hardware performance counters to obtain workload characteristics [12, 25, 105]. Some studies use cycle-accurate simulation tools to derive similar results [12, 41, 63, 90]. These studies often conclude either that two workloads show dissimilar behavior if their hardware performance counter characteristics are dissimilar, or that two workloads are similar if their hardware performance counter characteristics show similar behavior. In other words, two workloads are considered similar if they stress the microarchitecture of the processor in similar ways, and dissimilar otherwise.

Throughout the remainder of this chapter, we will use the most common workload characteristics measured using hardware performance counters: cycles-per-instruction (CPI), L1 data cache miss rate, L1 instruction cache miss rate, L2 cache miss rate, branch misprediction rate, data translation lookaside buffer (TLB) miss rate and instruction TLB miss rate. All miss rates are measured as the average number of misses per instruction. These characteristics are measured on an Intel Xeon L5420 system, which is an implementation of Intel's Core architecture on 45-nm technology. For more details on this system, we refer to Appendix C.1.1.

2.1.2 Pitfall in using hardware performance counters

There is a pitfall in most benchmark studies that rely heavily on workload characteristics obtained through hardware performance counters i.e., that the conclusions drawn from comparing the runtime behavior of programs using these metrics are potentially misleading. The fundamental reason for this pitfall is that different inherent (microarchitecture-independent) workload behavior can yield similar microarchitectural behavior. In other words, it is not because the performance achieved by a processor is similar in terms of metrics like CPI and cache miss rates for two different applications, that both workloads are stressing the processor in similar ways. In fact, the inherent behavior of the applications might be very different. The pitfall of

microarchitecture-dependent workload characterization is thus that the conclusions drawn from it may not be generalized to other microarchitectures [53, 54].

Quantifying the pitfall

In order to quantify this pitfall, we compare the differences that are observed between pairs of applications using two different workload characterization methodologies. For this, we characterize the benchmarks from five different benchmark suites, i.e., SPEC CPU2000, SPEC CPU2006, BioMetricsWorkload, BioPerf, MediaBench II.[1] We use the most common hardware performance counter metrics (CPI, cache miss rates, branch misprediction rate and TLB miss rates) and the set of microarchitecture-independent workload characteristics listed in Table 2.1. We will describe the microarchitecture-independent workload characteristics in more detail later.[2]

Using both types of workload characteristics, we build two normalized workload spaces. Each workload characteristic is normalized to a zero mean and unit standard deviation. This is done to put all characteristics on a common scale, regardless of the variation or the magnitude of the values.[3]

We then compute the Euclidean distance in both workload spaces between all pairs of workloads. Figure 2.1 shows the distance in the space constructed using normalized hardware performance counter (HPC) metrics on the vertical axis versus the distance in the normalized microarchitecture-independent (MICA) workload space on the horizontal axis. For convenience, we will refer to both workload spaces as the HPC space and the MICA space, respectively. Each dot corresponds to one pair of workloads.

There are several interesting observations to be made from Figure 2.1. First, it shows that two workloads may be very similar in terms of hardware performance counter metrics, while exhibiting significantly different microarchitecture-independent workload characteristics. This illustrates the pitfall in using hardware performance counter metrics in workload characterization studies clearly. It shows that although two workloads might exhibit similar behavior on one par-

[1] For more details on the benchmark suites used in this experiment, see Appendix B.
[2] See Section 2.1.3.
[3] See also the section on normalization in Section A.1.

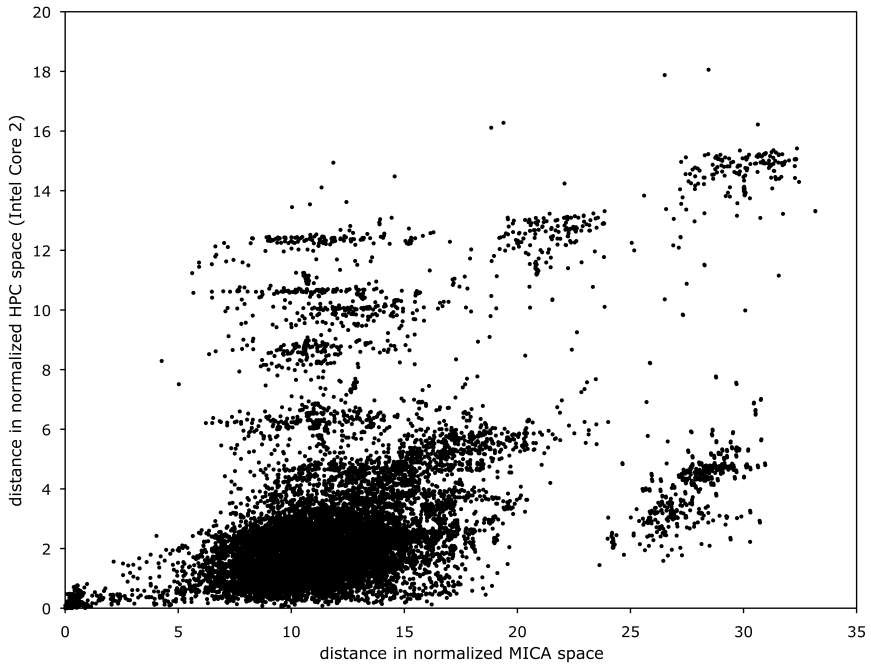


Figure 2.1: Illustration of the pitfall in using hardware performance counter based workload characterization. Each dot represents a pair of workloads; the vertical axis shows distances computed using hardware performance counter metrics collected on an Intel Xeon system, while the horizontal axis shows distance in terms of normalized microarchitecture-independent workload characteristics.

ticular microarchitecture, the inherent program behavior is potentially very different.

A second observation is that very few, if any, workload tuples show a very small distance in the MICA space and a large distance in the HPC space. This shows that the microarchitecture-independent workload characteristics are capable of capturing (dis)similarity in inherent program behavior: there are no pairs of workloads that are found to show similar microarchitecture-independent characteristics, but show dissimilar behavior in terms of hardware performance metrics.

It should be noted however that, once a certain threshold distance for a given workload tuple is observed in the MICA space, the workloads might show dissimilar hardware performance counter metrics. This is not surprising however. One example of a small difference in

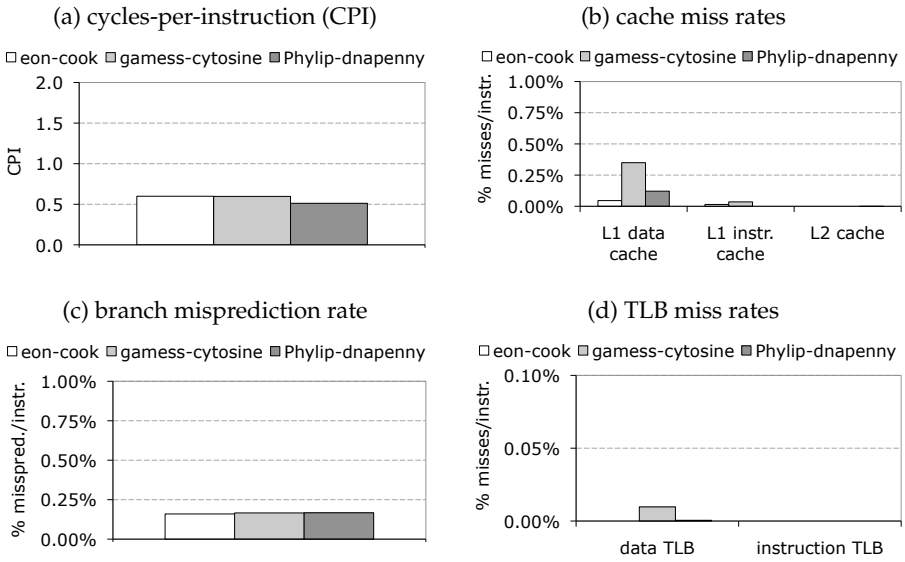


Figure 2.2: Hardware performance counter metrics measured on an Intel Xeon L5420 system (Core architecture), for three different workloads: SPEC CPU2000's eon with the reference input cook, SPEC CPU2006's gamess with the reference input cytosine, and BioPerf's Phylip with the medium input. The metrics suggest a high similarity of the runtime behavior of the three workloads.

program behavior that might result in large performance differences is branch predictability; even a seemingly small difference, e.g., a misprediction rate of 0.1% versus 0.5%, can potentially have a large impact on overall performance. Because this is captured by only a small portion of the microarchitecture-independent workload characteristics, this results in a relatively small distance in the MICA space. In terms of hardware performance counter metrics however, it is likely that a small difference in branch prediction accuracy is reflected by relatively large discrepancies for various metrics besides the branch misprediction rate, e.g., cycles-per-instruction (CPI) and potentially also L1 instruction cache miss rate, instruction TLB miss rate, etc.

Case study: eon vs gamess vs Phylip

We now further illustrate the pitfall in hardware performance counter workload characterization by means of a case study comparing the in-

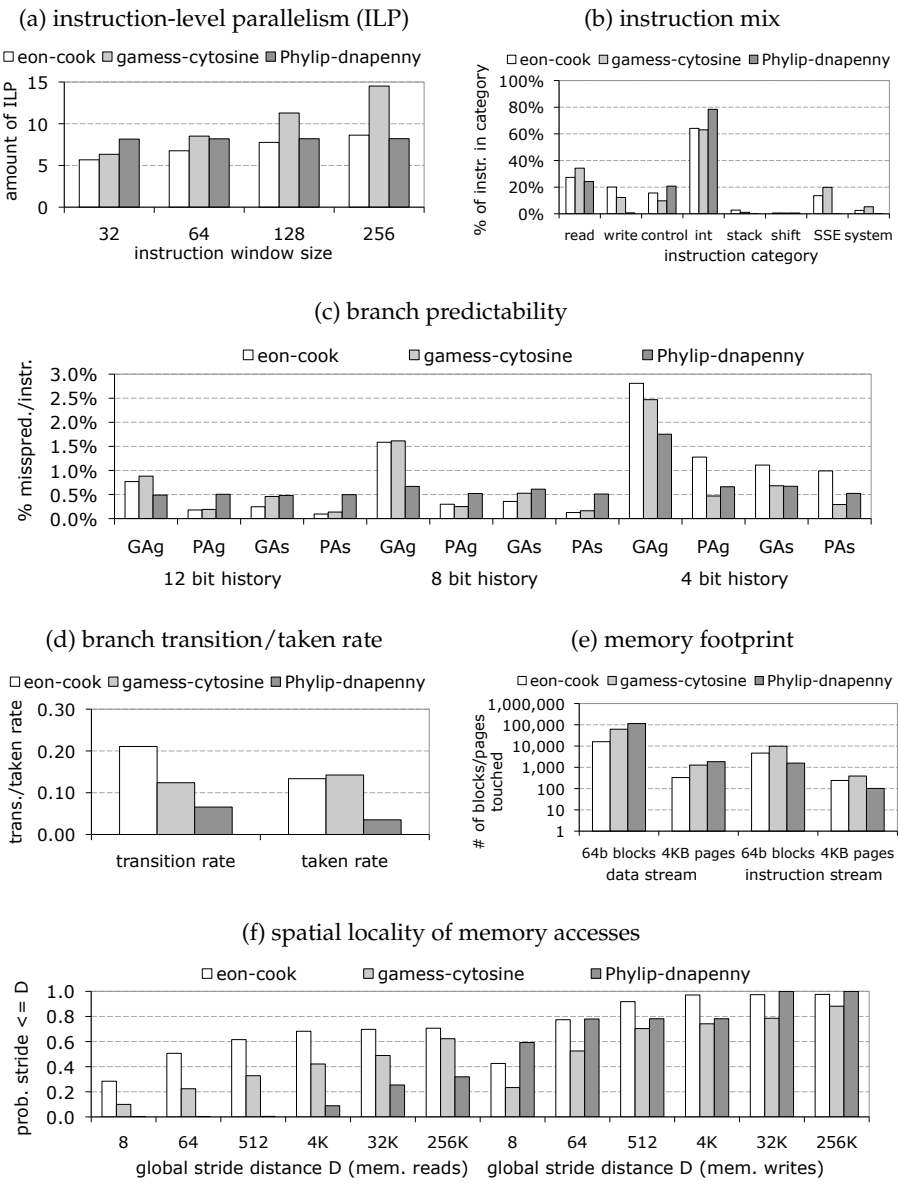


Figure 2.3: Microarchitecture-independent workload characteristics for three different workloads: SPEC CPU2000's eon with the reference input cook, SPEC CPU2006's gamess with the reference input cytosine, and BioPerf's Phylip with the medium input. Significant differences between the three workloads are observed in each group of characteristics.

herent behavior of three benchmarks, namely eon (SPEC CPU2000), gamess (SPEC CPU2006) and Phylip (BioPerf). Figures 2.2 and 2.3 show the hardware performance counter metrics and the microarchitecture-independent workload characteristics, respectively.

Figure 2.2 shows that the hardware performance counter metrics are very similar for eon, gamess and Phylip. However, the microarchitecture-independent workload characteristics shown in Figure 2.3 suggest that these workloads are fairly dissimilar. Significant differences are observed for each group of workload characteristics. Note that we omitted some workload characteristics from the graphs for brevity.

This example clearly illustrates that current practice in workload characterization, i.e., relying on hardware performance counter metrics, can be misleading. The set of microarchitecture-independent workload characteristics we present in the next section allows for comparing the true inherent behavior of workloads, instead of just the way in which the workloads stress one particular microarchitecture.

2.1.3 Microarchitecture-independent workload characterization

To capture the true inherent behavior of applications, we propose a set of microarchitecture-independent workload characteristics [54]. These are fundamentally different from the metrics described in Section 2.1.1, because they are not tied to the specific microarchitecture or processor. In other words, the values obtained for these workload characteristics will be the same regardless of the particular microarchitecture of the system on which they are measured, e.g., Intel NetBurst, Intel Core, Intel Nehalem, AMD Opteron, etc. Note that they are dependent on the instruction set architecture (ISA) being used (x86, x86-64, Alpha, ARM, etc.), and also on the particular compiler configuration used to build the application. In this work, we will measure these characteristics using the x86-64 ISA, for programs compiled with the GCC C/C++/Fortran compiler version 4.2.4, using the `-O2` optimization level.

In the following sections, we will describe each group of these workload characteristics in detail. Table 2.1 summarizes the set of 90 microarchitecture-independent workload characteristics. We found this set of workload characteristics sufficient to compare the inherent workload behavior for a large range of applications.

Table 2.1: Set of 90 microarchitecture-independent workload characteristics, divided in 7 groups: instruction mix, ILP, register traffic, branch predictability, memory footprint, temporal locality and spatial locality.

group	*no.*	*workload characteristic*
instruction mix [12]	1	fraction of instructions that read from memory
	2	fraction of instructions that write to memory
	3	fraction of control flow instructions
	4	fraction of integer instructions
	5	fraction of floating-point instruction
	6	fraction of stack operations (push/pop)
	7	fraction of shift operations
	8	fraction of string operations
	9	fraction of MMX/SSE instructions
	10	fraction of system operations (user-space)
	11	fraction of NOP operations
	12	fraction of other instructions
instruction-level parallelism (ILP) [4]	13	amount of ILP for instr. window of size 32
	14	amount of ILP for instr. window of size 64
	15	amount of ILP for instr. window of size 128
	16	amount of ILP for instr. window of size 256
register traffic [9]	17	average number of register input operands
	18	average degree of use
	19-25	prob. register dep. dist. $\leq 1, 2, 2^2, \ldots, 2^6$
branch predictability [14]	26-28	GAg PPM predictor miss rates (4,8,12 bits)
	29-31	GAs PPM predictor miss rates (4,8,12 bits)
	32-34	PAg PPM predictor miss rates (4,8,12 bits)
	35-37	PAs PPM predictor miss rates (4,8,12 bits)
	38	average branch taken rate
	39	average branch transition rate
memory footprint [4]	40	number of unique 64-byte blocks touched (data stream)
	41	number of unique 4 KB pages touched (data stream)
	42	number of unique 64-byte blocks touched (instr. stream)
	43	number of unique 4 KB pages touched (instr. stream)
temporal locality [28]	44-50	prob. global mem. read stride $\leq 0, 8, 8^2, \ldots, 8^6$
	51-57	prob. local mem. read stride $\leq 0, 8, 8^2, \ldots, 8^6$
	58-64	prob. global mem. write stride $\leq 0, 8, 8^2, \ldots, 8^6$
	65-71	prob. local mem. write stride $\leq 0, 8, 8^2, \ldots, 8^6$
spatial locality [19]	72	prob. cold memory read operations
	73-90	prob. LRU stack distance $\leq 2, 2^2, \ldots, 2^{18}$

Instruction mix

A first group of workload characteristics captures the instruction mix of the application. Each of these characteristics represents the ratio of the number of dynamically executed instructions of a particular instruction category to the total number of dynamically executed instructions.

We discriminate between 9 instruction categories: control flow instructions, integer instructions, floating-point instructions, stack operations, shift operations, string operations, MMX/SSE instructions, system operations and NOP operations. An extra group, referred to as *other*, is used to capture instructions that do not fit in any of these categories.

On top of this, we also determine the ratio of instructions that read from memory, and that write to memory. Because the x86-64 ISA is a CISC instruction set architecture (ISA), as opposed a RISC ISA (e.g., Alpha, ARM), this load/store aspect is not captured by the 10 other categories.

This results in 12 numbers which collectively characterize the dynamic instruction mix of an application.

Instruction-level parallelism

Instruction-level parallelism (ILP) quantifies the lack of dependences between instructions, and is defined as the average number of independent instructions over a fixed-size window of subsequent dynamic instructions. An instruction X is dependent on another instruction Y when X requires a result produced by Y, e.g., a value in a register or a value in a memory location. Note that the amount of available ILP is inherent to the particular workload being executed; the only microarchitecture-dependent factor is the size of the instruction window.

In order to measure the amount of ILP, we consider an out-of-order processor model in which every processor component is idealized and unlimited, except for the instruction window – we assume single-cycle latencies for caches (i.e., no cache misses), perfect branch prediction, an infinite number of functional units, etc. Using this idealized processor model, we measure the average amount of instructions-per-cycle (IPC) that can be achieved with a window size of 32, 64, 128 and 256 in-flight instructions.

The performance of a superscalar microprocessor is (in the absence of miss events) only affected by the number of available independent instructions and the fetch rate, and is limited by the issue width of the processor (typically 4 or 5). The number of independent instructions that can be executed in parallel is determined by the amount of available ILP. The importance of ILP with respect to the performance of a processor can be made apparent by applying Little's law, which states that the average number of tasks N in a system is equal to the frequency of incoming tasks λ times the average time required for handling a task T, i.e., $N = \lambda \cdot T$, or equivalently, $\lambda = N \ / \ T$. When applied to the performance of a microprocessor, we obtain IPC $= W \ / \ (K \cdot l)$, in which W is the number of instructions in the instruction window, K is the length of the critical path (assuming single-cycle latencies), and l is the average instruction latency. The critical path is defined as the longest path of dependent instructions present in the instruction window. Likewise, we can express the amount of ILP in terms of W and K, i.e., ILP$= W \ / \ K$. Thus, we obtain IPC $=$ ILP $/ \ l$, which clearly show the importance of the amount of ILP with respect to the performance of a processor.

Register traffic

We characterize the register traffic [40] between instructions using three metrics. First, we measure the average number of register input operands to an instruction. Next to this, we look at the average degree of use, i.e., the average number of times a register instance is consumed (register read) since its production (register write). Finally, we quantify the dependency distances between registers. The dependency distance is defined as the number of dynamic instructions between the production of a register and its consumption. This is captured by a set of probabilities, each of which reflect how often the register dependency distance is smaller than a value d, with $d = 2^i$ and i in $\{0, 1, 2, \ldots, 7\}$.

Branch predictability

Branch predictors are a very important component in modern microprocessors. By predicting the direction (taken, not-taken) and/or target of a particular branch with very high accuracy, they form a crucial component of an out-of-order microprocessor in order to achieve high performance [106].

In order to characterize the predictability of branch behavior independently of a particular microarchitecture, we use the Prediction by Partial Matching (PPM) predictor [23], a universal compression/prediction technique. We can view the PPM predictor as a theoretical basis for branch prediction, rather than an actual predictor that is to be built in hardware.

We will consider four variations of the PPM predictor: GAg, PAg, GAs and PAs. 'G' means global branch history whereas 'P' stands for per-address or local branch history; 'g' means that one global predictor table is shared by all branches, while 's' indicates separate tables per branch. For each variation, we consider three different maximum history lengths, i.e., 4, 8 and 12 bits.

In addition, we also quantify the average branch taken rate and average branch transition rate. The branch taken rate is simply the percentage of times a conditional branch was taken. The branch transition rate quantifies how frequently the direction of a conditional branch changes from taken to non-taken (or vice versa) dynamically [50]. Conditional branches that are highly biased towards one direction or that show a very low or very high transition rate tend to be easier to predict.

Memory footprint

The memory footprint of a workload is defined as the amount of memory that the application uses during its execution. We characterize this as the number of unique 64-byte blocks that were touched together with the number of unique 4 KB pages that were touched. We count the number of blocks and pages separately for accesses to data (data stream) and accesses to instructions (instruction stream).

Spatial locality of memory accesses

To characterize the spatial locality of memory accesses of a workload, i.e., the patterns in which the workload accesses memory, we look at local and global data strides. A global stride is defined as the difference in the data memory addresses between temporally adjacent memory accesses. A local stride is defined in the same way, except that both memory accesses come from a single static instruction – this is done by tracking memory addresses for each memory operation.

The stride behavior is characterized by a set of probabilities that

reflect how often a stride was smaller than or equal to 0 and 8^i, with i in $\{1, 2, \ldots, 6\}$. We make a distinction between the stride behavior of memory reads and memory writes.

Because these workload characteristics characterize the spatial locality of memory accesses, they are particularly important with respect to cache behavior and prefetching.

Temporal locality of memory accesses

Next to characterizing spatial locality, it is equally important to quantify the temporal locality of memory accesses, which can be characterized by the number of accesses to unique memory addresses between two accesses to the same memory address. High temporal locality suggests good cache behavior, whereas poor temporal locality is a possible cause of a high cache miss ratio. Since temporal locality of memory read operations is more important than that of memory writes with regard to performance, since the latter are typically buffered and do not necessarily stall the processor pipeline, we only characterize the former.

To characterize the temporal locality of the memory reads performed by a workload we look at the distribution of least-recently-used (LRU) stack distances [18]. The LRU stack distance of a memory read operation is defined as the number of unique memory read operations between two reads of the same memory address. A small LRU stack distance corresponds to good temporal locality, while a large distance is an indication of poor temporal locality. We consider 64-byte blocks instead of individual memory addresses for efficiency.

Again, we use a set of probabilities to capture the distribution of values. Each probability reflects how often an LRU stack distance smaller than a given value was observed. We collect probabilities for distances smaller than 2^i, with i in $\{1, 2, \ldots, 18\}$. In addition, we also keep track of the frequency of cold memory read operations, i.e., read accesses to addresses not accessed before, and that of LRU stack distances greater than 2^{18}.

MICA Pin tool

To collect these microarchitecture-independent workload characteristics, we developed a Pin tool called MICA (Microarchitecture-Indepen-

dent Characterization of Applications) [4]. Pin is a binary instrumentation framework provided by Intel [5], which allows to easily implement analysis tools. A Pin tool can be used as a plug-in to the Pin framework, which provides a rich API for analyzing the runtime behavior of a program. MICA is also being used by other research groups, including researchers at the University of Virginia, University of California and Simon Fraser University (Canada) [22, 75, 91, 112].

Depending on the analysis being done, instrumentation may introduce a significant overhead, resulting in substantial slowdowns compared to native execution. The most expensive analysis implemented in MICA is collecting the ILP workload characteristics, which results in a slowdown of up to $600\times$. This is not surprising, keeping in mind that the ILP analysis is done by actually simulating an idealized out-of-order processor. Measuring the other groups of microarchitecture-independent workload characteristics is less intrusive, ranging from a slowdown of roughly $200\times$ to $10\times$ slowdown. Note that the slowdowns observed when simulating workloads using a state-of-the-art cycle-accurate simulator are substantially higher; slowdowns of at least four orders of magnitude (10,000$\times$) are common [24].

2.2 Phase-level workload characterization

Most workload characterization studies limit themselves to aggregate workload characterization [51, 86, 88], i.e., they report and analyze workload characteristics that represent the average behavior of workloads across the entire program execution. However, only relying on aggregate workload characterization might, again, be misleading.

Consider for example the case where a workload characterization study would report that, for a given workload, 30% of the instructions executed read from memory. This would make a computer architect conclude that about one third of the functional units being load/store units would suffice for this workload to achieve good performance. However, during the first half of program execution, only 10% of the instructions may be reading from memory; and during the second half, there may be 50% instructions in the dynamic instruction stream that need to read from memory. On average over the entire program execution, this results in 30% instructions reading from memory. Obviously,

[4] MICA is available at `http://www.elis.ugent.be/~kehoste/mica`

[5] `http://www.pintool.org`

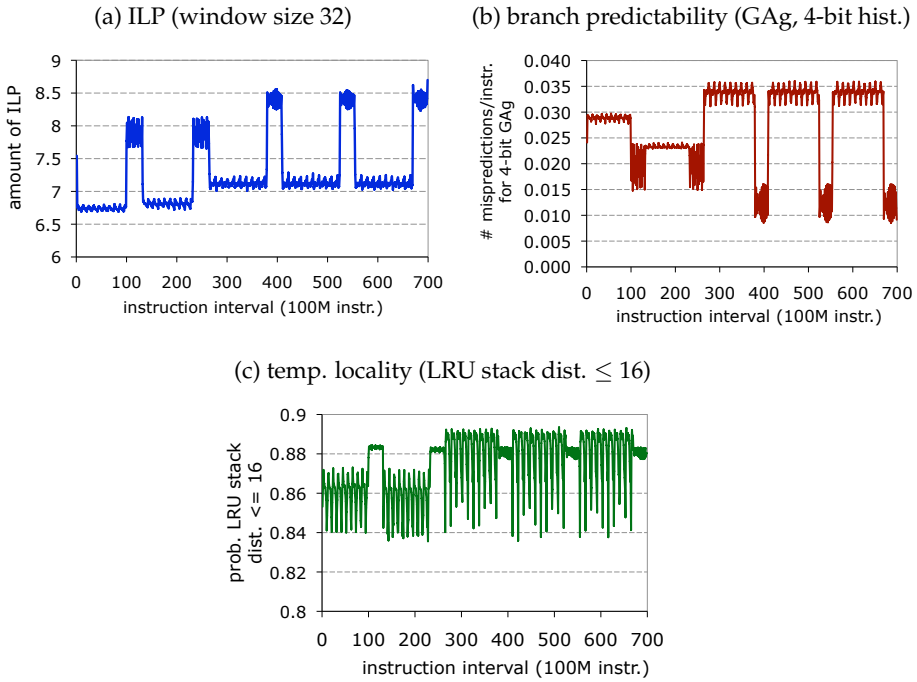


Figure 2.4: Illustration of phase behavior of gzip-graphic. The three subfigures show the variation for three different microarchitecture-independent workload characteristics over time, i.e., (a) the amount of available ILP for an instruction window size of 32, (b) the branch predictability for a GAg PPM predictor with 4-bit history, and (c) the temporal locality expressed as the probability of the LRU stack distance being smaller than 16 64-byte blocks.

one third of the functional units being load/store units (based on the aggregate analysis) would yield good performance for the first part of the program execution, but would yield suboptimal and unexpectedly low performance for the second part. A phase-level characterization showing that there are two major program phases each exhibiting different behavioral characteristics would be more accurate and more informative.

In this section, we present a feasible way of doing phase-level workload characterization studies. First, we compare aggregate and phase-level workload characterization in Section 2.2.1 and discuss the challenges of a phase-level workload characterization study in Section 2.2.2. Our methodology is outlined in Section 2.2.3.

2.2.1 Aggregate versus phase-level workload characterization

A phase is (informally) defined as an interval of temporally adjacent dynamic instructions in which the behavior of the workload is more or less homogeneous with respect to one or more workload characteristics. In this work, we collect workload characteristics for fixed intervals of 100 million (10^8) dynamic instructions. This allows us to capture the phase-behavior of workload executions, while keeping the amount of data that needs to be processed more or less manageable. Also, intervals of this order of magnitude are large enough to avoid warmup issues in simulation studies [84], which further motivates our choice.

Figure 2.4 shows the time-varying behavior of the SPEC CPU2000 gzip benchmark with the graphic reference input in terms of ILP, branch predictability (4-bit history GAg PPM predictor) and temporal locality, i.e., the probability of the memory access LRU stack distance to be smaller than 16 64-byte blocks. Just reporting average values for these workload characteristics, i.e., 7.265, 0.028 and 0.878 respectively, completely hides the phase behavior of the workload that is clearly visible when the values of the different workload characteristics are plotted as a function of time.

2.2.2 Challenges in phase-level workload characterization

Although the notion of time-varying behavior is well known to computer architects, and although there is a lot of recent work on detecting and exploiting phase behavior [94, 95], most workload characterization work is still limited to aggregate workload analysis and does not study time-varying behavior.

The motivation behind this is straightforward: aggregate analysis requires no new methodology for processing and presenting the data compared to previous studies. Even if the number of workloads being studied is large (e.g., tens to hundreds), and a large number of different workload characteristics are being used, the amount of data that needs to be processed and handled is relatively small. Thus, relying on traditional ways to present the data in a concise manner, e.g., using multiple bar plots and/or scatter plots, is often sufficient.

In case of phase-level workload characterization on the other hand, one quickly obtains a very large data set, up to the point where traditional approaches for handling and processing the data becomes in-

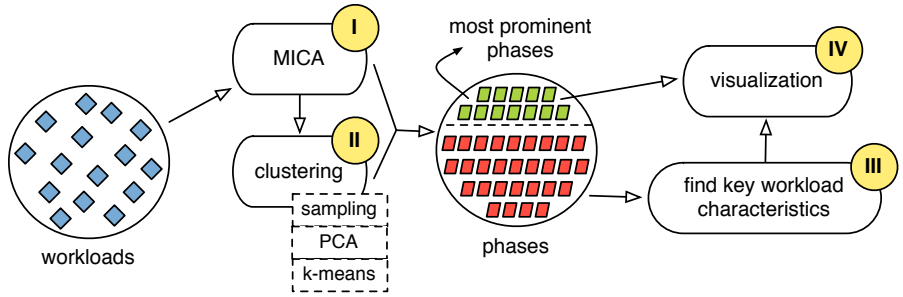


Figure 2.5: Outline of the proposed phase-level workload characterization methodology, which consists of 5 steps: microarchitecture-independent workload characterization, sampling, clustering, finding key workload characteristics and visualization of the most prominent phase behaviors.

tractable. For example, the data set which will be used in Section 2.3 consists of workload characteristics for 100 M instruction intervals of 77 benchmarks. Together, this results in 90 microarchitecture-independent workload characteristics for over one million instruction intervals. It is obvious that getting insight into this huge data set is non-trivial.

2.2.3 Phase-level workload characterization

In this section, we present a feasible phase-level workload characterization methodology that uses microarchitecture-independent workload characteristics. After finding the most prominent phase behaviors in terms of these workload characteristics, we identify a subset of key characteristics which are then used to visualize these phase behaviors.

The methodology is discussed in general here, and is applied in the next section to analyze a large data set of fine-grained workload characteristics. The steps are outlined in detail in the following paragraphs and are illustrated in Figure 2.5. Note that although these steps are targeted towards analyzing the most prominent phase-level behavior of a set of applications, they could be easily adopted to other purposes, e.g., finding extreme behavior or phase behavior which is common across multiple workloads or programs.

Step I: Microarchitecture-independent workload characterization

In the first step, we collect the microarchitecture-independent workload characteristics presented in Section 2.1.3 per execution interval of 100 million (10^8) dynamic instructions. The following steps are intended to cope with the large amount of data obtained.

Step II: Finding the most prominent phase behaviors

The goal of the second step in the methodology is to find the most prominent phase behaviors. We use the data set obtained in step I, normalized to a zero mean and unit standard deviation per microarchitecture-independent workload characteristic. This is done to ensure equal weight for each workload characteristic, regardless of the variation or range of values.

Finding the most prominent phase behaviors is done in a series of substeps, as indicated in Figure 2.5. We subsequently sample the huge data set per benchmark, apply Principal Component Analysis (PCA) to obtain a small number of uncorrelated underlying dimensions, and then perform cluster analysis to find groups of similar execution intervals.

Sampling The first of these substeps is intended to cope with the huge amount of data, by significantly reducing the size of the data set by means of randomly sampling execution intervals per benchmark. Because workloads often exhibit distinct phase behavior (see Figure 2.4), which results in large subsets of very similar execution intervals, this step causes very little information loss. It does however significantly reduce the resources required to perform the following steps of the methodology.

We randomly select a fixed number of intervals per program (or benchmark), as opposed to sampling per workload, i.e., a program run with a particular input set. For programs for which fewer intervals are available in the data set, this means that instruction intervals will appear multiple times in the final sampled data set. The motivation behind this is that the data set is likely to contain workloads which show a significantly longer runtime than others, or multiple workloads that correspond to the same program being run with different input sets. By sampling per program, we ensure equal weight for each program

in the subsequent steps of the methodology. This is important, because otherwise long-running applications or programs with multiple inputs would indirectly get more emphasis in PCA and clustering. Note that this is a design choice: if equal weights per workload, or per group of workloads (e.g., benchmark suite) are preferred, this step can be adjusted accordingly.

Principal Component Analysis Applying PCA prior to performing cluster analysis is important for several reasons. For one, by only retaining the p most significant principal components, we emphasize on the underlying dimensions in the data set along which the largest variation is observed. Removing dimensions which contribute less to the overall variance thus not only reduces the dimensionality of the data set on which cluster analysis will be done, which will benefit the time needed to perform the clustering; it also contributes to focusing on large differences in phase behavior. A second major reason to apply PCA prior to clustering is the fact that principal components are uncorrelated. Computing a distance across uncorrelated dimensions is clearly preferred over computing a distance across potentially highly correlated dimensions. Normalizing the retained principal components is required to put them on a common scale. For a thorough overview of PCA, we refer to Appendix A.1.

Cluster analysis Cluster analysis is performed using the k-means clustering algorithm [64], which is an iterative process that first randomly selects k cluster centers, and then continues in two steps per iteration. The first step is to compute the Euclidean distance of each point in the multidimensional space to each cluster center. In the second step, each point gets assigned to the closest cluster. As such, new cluster centers are to be computed. The algorithm iterates until convergence is observed, i.e., until cluster membership ceases to change across iterations (or until a maximum number of iterations is reached). K-means clustering yields spherical clusters, i.e., sets of data points located around a cluster center. In the context of clustering execution intervals, each cluster represents a program phase, i.e., a set of intervals with similar workload characteristics. For a particular cluster, the interval closest to the cluster center can be used as a representative for all intervals in that cluster, thus serving as an example for the corresponding phase.

The final goal of this clustering step is to yield a limited number of representative execution intervals that collectively cover a sufficiently large fraction of the entire data set of benchmarks. This involves making a trade-off between coverage and variability within a cluster. To illustrate this, consider the following example. Say the ultimate goal for the workload characterization study is to come up with 100 prominent phases. One option would be to apply k-means clustering with k set to 100; this will yield 100 prominent phases with a 100% coverage, i.e., all instruction intervals in the data set will be represented by a phase representative. Another option is to apply k-means clustering for $k \gg 100$; in this case, the 100 most prominent phases, i.e., the 100 largest clusters, will account for less than 100% coverage. However, the variability within each cluster will be substantially smaller than for the $k = 100$ case. In other words, we can select the most prominent phase behaviors that collectively account for a large fraction of the entire benchmark suite while minimizing the variability of each prominent phase.

Note that for identifying the most prominent phases, we map the full data set obtained in step I to the clusters obtained with k-means. We only sampled the data set prior to cluster analysis to speed up this step, and to assure an equal weight for each benchmark in the clustering analysis. However, when evaluating how prominent the phase behavior represented by each particular cluster is, we should also take the non-sampled execution intervals into account. This is also important for subsequent analysis of the uniqueness and diversity of the programs in terms of phase behavior (see Section 2.3.2).

This step of the overall methodology again gives the analyst some freedom: retaining more or less principal components will result in a more or less fine-grained analysis, while changing the value of k and the number of prominent phases retained for subsequent analysis allows trading off coverage and in-phase variability.

Step III: Identifying key workload characteristics

In order to ease gaining insight into a large data set, reducing the dimensionality of the data is a logical step: interpreting data that is expressed using just N features is significantly easier than interpreting an equivalent data set with M features, with $N \ll M$. Limiting the number of workload characteristics without losing too much information will not only help to understand the inherent behavior of the workloads but could also limit the time required to collect additional data.

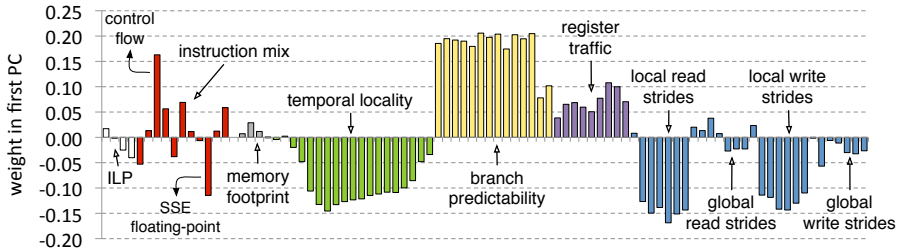


Figure 2.6: Factor loadings for the first principal component in a phase-level workload characterization study using microarchitecture-independent workload characteristics. Giving a meaningful interpretation to the underlying dimension represented by this PC is not straightforward.

This is especially important when the data is being collected with techniques which suffer from significant slowdowns compared to native execution, e.g., simulation or instrumentation.

In the machine learning community, there is a large body of work on so-called *feature selection* techniques [48], which lower the dimensionality of a data set in various ways. Each technique has its strengths and weaknesses with regard to time complexity, heuristical or exhaustive nature, interpretability of the results, etc. Here, we will consider three such techniques in the context of workload characterization.

Principal Component Analysis In step II, we already relied on Principal Component Analysis to capture the major trends in the data set and make the clustering of execution intervals faster. PCA has also proven its value in other workload characterization studies involving the construction of a subset of representative workloads [108].

However, although principal components are just linear combinations of the original input variables, in this case microarchitecture-independent workload characteristics, it is sometimes hard to give a meaningful interpretation to them. This is illustrated in Figure 2.6, which shows the factor loadings obtained in step II, for the first principal component retained from the data set being studied in Section 2.3.

Interpreting just this first principal component which represents the most prominent underlying dimension is already surprisingly difficult; the best we can come up with is that this underlying dimension differentiates between workloads with difficult to predict control-intensive

behavior and floating-point workloads[6] with fairly high temporal locality and very good spatial locality (both for memory reads and writes). It is clear that such rather complex interpretations for each of the principal components will make a comparison of the inherent behavior of workloads difficult. Furthermore, a value close to zero along a principal component can mean one of two things. Either all the workload characteristics that have a significant factor loading in that principal component are close to zero, or the values of workload characteristics with a significantly high or low factor loading balance each other out.

The goal of representing the workloads using just a small subset of workload characteristics is to make it easier to compare the inherent behavior of those workloads. In our opinion, this goal is not met when using PCA for identifying key workload characteristics, because of the difficulties described above with interpreting the principal components. Therefore, we also consider two other feature selection techniques for selecting subsets of the original workload characteristics.

Correlation elimination Correlation elimination is a so called *greedy backward elimination* feature selection technique [48]. It is classified as a backward elimination technique because it selects a workload characteristic for removal in each iteration, starting from the set including all workload characteristics. The greedy aspect is reflected in the way in which it selects the next characteristic for removal; it will select the characteristic which causes the smallest drop in the overall evaluation criterion, without reconsidering previous choices.

This technique heavily relies on the correlation between the different workload characteristics to construct a subset. It uses an iterative algorithm, and works as follows. For each workload characteristic w_i, the Pearson correlation coefficient ρ_j [64] with all the other workload characteristics w_j $(1 \leq j \leq N, i \neq j)$ is computed, and then combined into an average correlation coefficient $\overline{\rho_i}$ as follows:

$$\overline{\rho_i} = \frac{1}{N} \sum_{i \neq j} \rho_j$$

All workload characteristics are then ranked by their corresponding average correlation coefficient $\overline{\rho_i}$. The workload characteristic that

[6] For x86-64, GCC emits SSE instructions for all floating-point work by default, because these instructions are more effective than the X87 floating-point instructions used on platforms which do not support SSE, as is the case for some 32-bit x86 platforms.

shows the highest average correlation coefficient is then removed, by which we implicitly assume that it contains the least additional information compared to all the other workload characteristics. This results in an $(N-1)$-dimensional data set.

This process is iterated by progressively removing additional workload characteristics, until a predetermined number of workload characteristics is retained. By eliminating highly correlated characteristics, we reduce the dimensionality of the data set without losing the insight that the workload characterization provides.

Although this technique avoids the interpretation issues observed with PCA, it is likely to yield significantly suboptimal sets of key workload characteristics because of its greedy nature. Greedy algorithms are known to be very sensitive to getting stuck in local optima of the search space. Unfortunately, evaluating all possible subsets of our set of 90 microarchitecture-independent workload characteristics as key workload characteristics is out of the question. Even with a very fast evaluation function and using huge computing resources, the search space of 2^{90} (roughly 10^{27}) subsets is simply too large. Therefore, we consider one last feature selection technique which avoids the downsides of both PCA and correlation elimination.

Genetic algorithm By building a specialized feature selection technique which relies on a genetic algorithm, which is a broadly applied technique for quickly finding adequate solutions in a huge search space, we are able to obtain a set of original key workload characteristics, thus avoiding the interpretation concerns of principal components. Because of its evolutionary rather than greedy nature, this technique will quickly focus on the most important workload characteristics required for capturing the major differences between the inherent behavior of the workloads. For a detailed discussion on genetic algorithms, see Section A.2.

For the purpose of identifying key workload characteristics, we define a candidate solution represented by an entity as a vector of N 0's and 1's, with N being the number of workload characteristics ($N = 90$ using our set of microarchitecture-independent workload characteristics). The inclusion of a workload characteristic in a particular candidate solution is then indicated by a 1 in the vector, while a 0 indicates exclusion of a workload characteristic.

The fitness function $f(e)$ used to compute the fitness of an entity e

for this particular problem consists of two factors:

$$f(e) = \rho_e \cdot (1 - \frac{n_e}{N})$$

The first factor ρ_e corresponds to the Pearson correlation coefficient of the Euclidean distances between the benchmark tuples using the original set of workload characteristics with the distances between those tuples using the subset of workload characteristics described by e. This will cause the genetic algorithm to try and find a subset of workload characteristics that correlates well with the original set in terms of the differences that are observed between benchmark tuples. This is a direct translation of the desire to obtain a set of key workload characteristics for comparing the inherent program behavior of a set of workloads. As before, the Euclidean distances are calculated in terms of normalized principal components, which are uncorrelated; for both sets of workload characteristics we retain all principal components with a standard deviation greater than one.

The second factor $(1 - \frac{n_e}{N})$, in which n_e is the number of selected workload characteristics in entity e (i.e., the number of 1's), is a simple but effective heuristic which rewards subsets with fewer selected characteristics. The purpose of this factor is to make the genetic algorithm focus more on smaller subsets of workload characteristics, if those are able to achieve a sufficiently high correlation coefficient ρ_e.

Step IV: Visualizing the prominent phase-level workload behavior

The last step of the methodology consists of visualizing the most prominent phase behaviors using the key microarchitecture-independent workload characteristics. Because we prefer a concise way of visually representing the different key workload characteristics of the potentially large number of retained phases, we opt for so-called *kiviat diagrams*.

A kiviat diagram, also known as a radar plot, represents one of the prominent phase behaviors (see Figure 2.7). The values of each workload characteristic are plotted as points on axes organized in a circular way, and are then connected pairwise to form a single 2-dimensional area which represents the phase behavior in terms of the key workload characteristics. Each of the axes are divided in four ranges, using the mean, the mean minus one standard deviation and the mean plus one

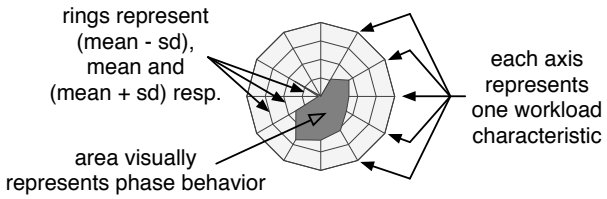


Figure 2.7: Legend for a kiviat diagram representing a prominent phase behavior.

standard deviation as boundary values. This way, extreme phase behavior is easier to recognize when comparing the kiviat diagrams.

2.3 Application: Comparing phase-level workload behavior across benchmark suites

We now apply the methodology to a data set of microarchitecture-independent workload characteristics for over one million execution intervals, which represent the inherent program behavior of 77 benchmarks taken from 5 different benchmark suites, i.e., BioMetricsWorkload (biometrics), BioPerf (bioinformatics), MediaBench II (multimedia), and SPEC CPU2000 and CPU2006 (general-purpose computing). For a detailed overview of these benchmark suites, see Appendix B.

The goal of this workload characterization study is to compare these benchmark suites in terms of time-varying inherent program behavior, and also evaluate the uniqueness and diversity of the benchmarks, which can be done by applying the methodology presented in the previous section.

2.3.1 Applying the methodology

The following sections detail on the application of the various substeps of the presented methodology to our data set.

Finding the most prominent phase behaviors

After applying step I of the methodology, we obtain 90 microarchitecture-independent workload characteristics for each of the 1,090,005 100 M-instruction execution intervals. In the second step, we randomly

Figure 2.8: Cluster coverage of the 100 largest clusters for different numbers of clusters.

select 1,000 execution intervals per benchmark. Collectively, this yields a data set of just 77,000 intervals, a reduction of over 90% in terms of the amount of data that needs to be processed in the subsequent substeps of step II of the methodology. For most benchmarks more than 1,000 execution intervals are available in the full data set, so only a few benchmarks will have intervals appearing multiple times in the sampled data set. This is a small sacrifice we make to ensure that all benchmarks in the PCA and cluster analysis all get an equal weight, which we believe is important in a workload characterization study. Prior to the cluster analysis in step II, we retain all principal components which show a standard deviation of one or higher. This results in 19 principal components, which collectively contain 84.81% of the total variance observed in the 90 original microarchitecture-independent workload characteristics. Thus, we will obtain relatively coarse-grained prominent phase behaviors.

Figure 2.8 shows the coverage of the 100 largest clusters obtained with k-means clustering, for different values of the total number of clusters k. The largest 100 out of 300 clusters obtained by applying k-means clustering cover 89.61% of the entire set of 100 M-instruction execution intervals, which matches our preset goal. Figure 2.9 shows the mean in-cluster variance for the same cluster configurations. Clustering to more than 300 clusters only marginally improves the mean in-cluster variability relative to the loss in coverage, further motivating this selected setting. We will study the clustering of the execution intervals into these 300 clusters in detail in Section 2.3.2.

Figure 2.9: Mean in-cluster variance for different numbers of clusters.

Identifying the key workload characteristics

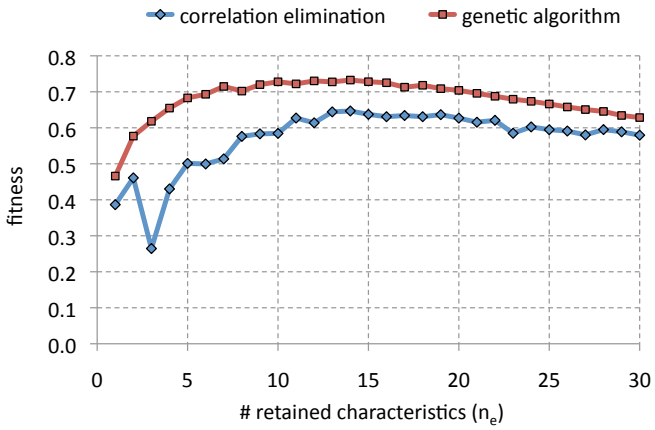
The next step in the methodology is to identify a set of key microarchitecture-independent workload characteristics. The genetic algorithm was configured to evolve a single population of 500 entities with an archive of 100 entities, using a crossover rate of 85% and a mutation rate of 15%. For crossover, we use crossover mixing with a mixing rate of 25%, while mutation is done using multi-point drift with a control parameter value of 0.25[7]. The algorithm reached convergence in just 32 generations or less for each number of selected workload characteristics. We found these empirically determined settings to yield good results, both in terms of the quality of solutions and the required exploration time. We ran the genetic algorithm multiple times, each time for a fixed number of workload characteristics to select. That way we can trade off the distance correlation ρ_e and the number of retained characteristics, and also easily compare the quality of the key workload characteristics with those obtained using the correlation elimination technique.

Figure 2.10 shows the quality of the sets of key workload characteristics obtained using both correlation elimination and the genetic algorithm, in terms of the fitness score as defined in Section 2.2.3 and the Pearson correlation coefficient ρ_e of the distances between all pairs of phase representatives.

These results support the claims in Section 2.2.3 about the greedy nature of the correlation elimination technique. The pair-wise distances

[7] See Section A.2.

(a) fitness = $\rho_e \cdot \frac{1 - n_e}{N}$



(b) Pearson corr. coef. (ρ_e)

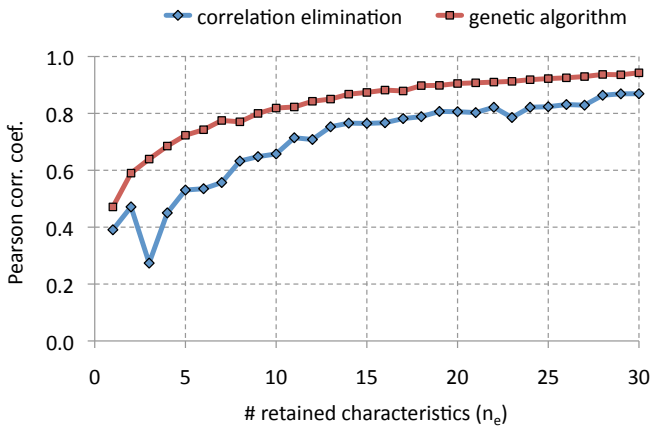


Figure 2.10: Comparison of the sets of key workload characteristics obtained with correlation elimination and the genetic algorithm. The fitness score for 1 up to 30 retained key workload characteristics is shown on the top graph, the Pearson correlation coefficient is shown in the bottom graph.

Table 2.2: Set of 10 key workload characteristics selected by the genetic algorithm, with a correlation coefficient of the pair-wise distances between phase representatives of 0.8188.

id	*label*	*key workload characteristic*
k_1	`ilp_64`	amount of ILP for instr. window of size 64
k_2	`ratio_string`	ratio of string instructions
k_3	`ratio_nop`	ratio of NOP instructions
k_4	`instr_ftprnt_64b`	number of unique 64-byte blocks touched (instruction stream)
k_5	`prob_cold_mem`	prob. cold memory read operations
k_6	`prob_LRU_dist_lt_2`	prob. LRU stack distance ≤ 2
k_7	`prob_LRU_dist_lt_8k`	prob. LRU stack distance $\leq 2^{13}$
k_8	`prob_lw_stride_lt_64`	prob. local mem. write stride $\leq 8^2$
k_9	`prob_lw_stride_lt_256k`	prob. local mem. write stride $\leq 8^6$
k_{10}	`prob_gw_stride_eq_0`	prob. global mem. write stride ≤ 0

obtained using the sets of key workload characteristics retained with correlation elimination show significantly lower correlation coefficients with the distances obtained using the full set of workload characteristics compared to those obtained with the genetic algorithm (see Figure 2.10b). Naturally, this also results in significantly lower fitness scores when the factor which rewards smaller sets of key workload characteristics is taken into account.

Another interesting observation is the effectiveness of the reward factor in the fitness score. Figure 2.10b shows that the distance correlation ρ_e ramps up fairly quickly initially, as more key workload characteristics are included, but then starts flattening beyond 10 key workload characteristics. The fitness score shown in Figure 2.10a captures this well; the set of 10 key workload characteristics obtained using the genetic algorithm yields a fitness score very close to the highest fitness score, and including more characteristics beyond 15 only lowers the fitness, since the additional gain in the correlation of the distances does not outweigh the larger number of selected characteristics.

Thus, we retain the 10 workload characteristics identified using the genetic algorithm in step IV of the methodology, which results in a correlation coefficient ρ_e of 0.8188. Table 2.2 lists the selected key microarchitecture-independent workload characteristics. Next to the amount of ILP ($k1$) , the ratio of string ($k2$) and NOP instructions ($k3$) and the instruction memory footprint ($k4$), a large part of the key workload

characteristics concern the temporal locality of memory read accesses ($k5 - k7$) and the spatial locality of memory write accesses ($k8 - k10$), which suggests the memory access patterns of the 77 workloads studied are quite different.

Visualizing the prominent phase-level workload behavior

The final step of the methodology is to use this set of key workload characteristics to visualize the 100 most prominent phase behaviors using kiviat diagrams (see Figures 2.11, 2.12 and 2.13). We organize the kiviat diagrams in three groups. We make a distinction between *benchmark-specific* phase behaviors which stem from a single benchmark, *suite-specific* phase behaviors which stem from multiple benchmarks from a single benchmark suite, and *mixed* phase behaviors which stem from different benchmarks from multiple benchmark suites.

Each kiviat diagram is accompanied with a legend specifying the relevant benchmarks for the presented phase behavior and the percentage of their total runtime for which the particular phase behavior is representative. The pie chart indicates the weight of each benchmark in the prominent phase behavior. For example, the leftmost kiviat diagram in the first row of Figure 2.13 shows the phase behavior that is present in both the wrf and zeusmp benchmarks. About 10% of the execution intervals represented by this phase behavior stem from the wrf benchmark, and capture 2.72% of the runtime behavior of wrf; the remaining intervals stem from the zeusmp benchmark, and collectively represent almost 66% of the runtime behavior of that benchmark.

Discussion

Several interesting observations can be made from these kiviat diagrams when comparing the most significant phase behaviors.

Unique phase behavior The benchmark-specific kiviat diagrams represent unique phase behaviors not observed in other benchmarks. The BioPerf, SPECint2006 and SPECfp2006 benchmark suites, and to a lesser extent also the SPECint2000, SPECfp2000 and BioMetricsWorkload benchmark suites, exhibit a number of unique phase behaviors (see Figures 2.11 and 2.12), and the kiviat plots provide insight into why these behaviors are unique. For example, the runtime behavior

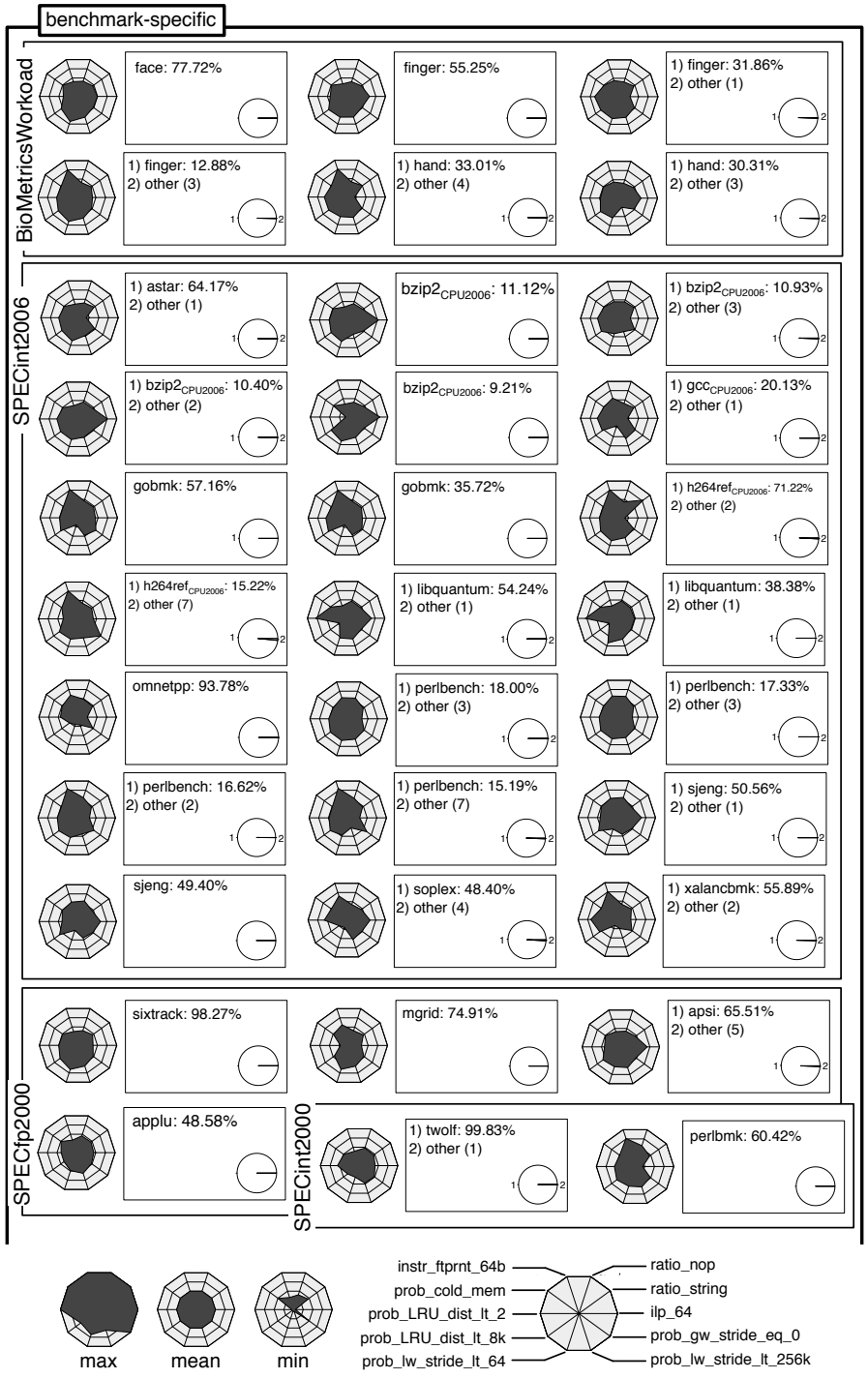


Figure 2.11: Kiviat plots (part I) representing the prominent phase behaviors.

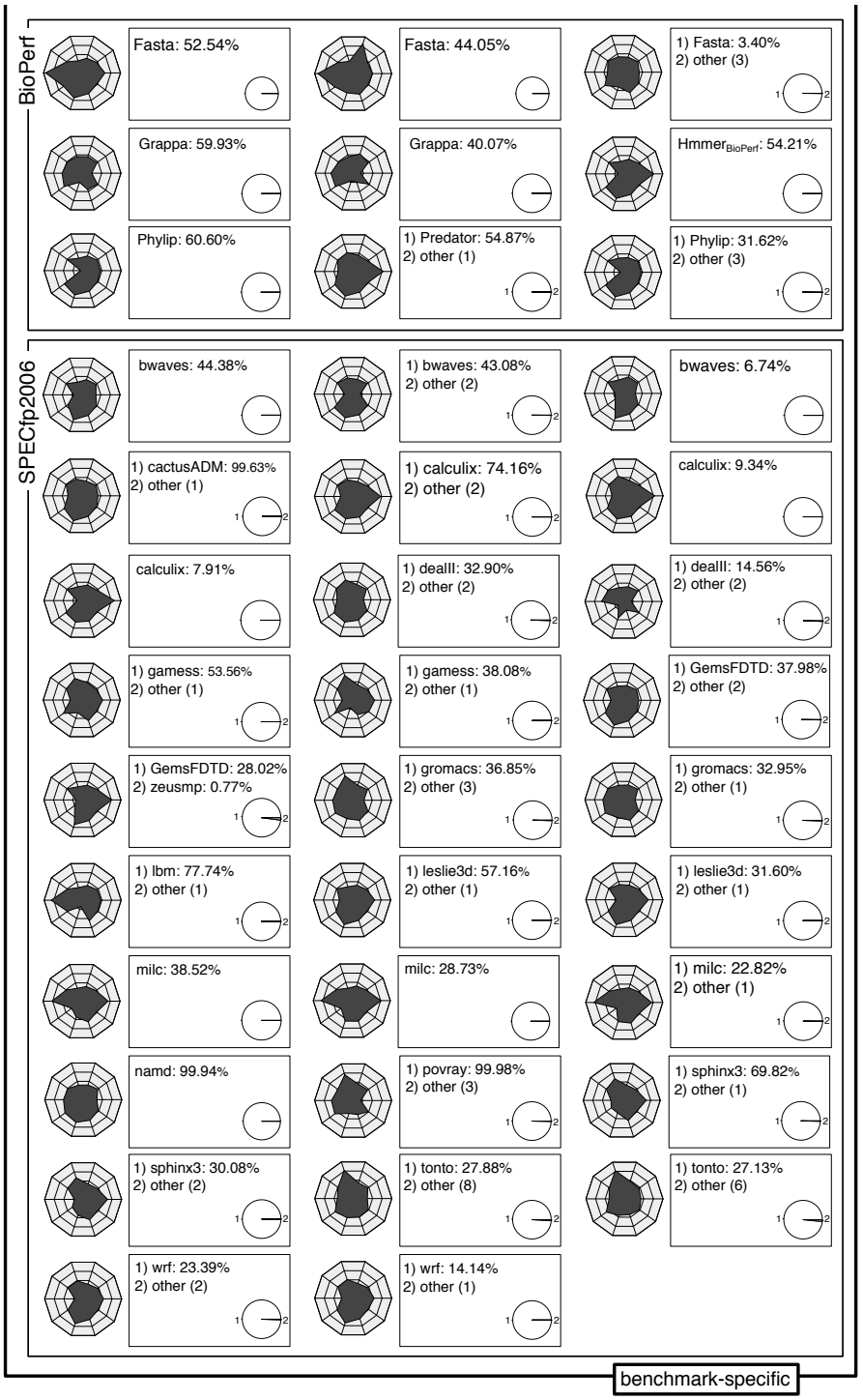


Figure 2.12: Kiviat plots (part II) representing the prominent phase behaviors.

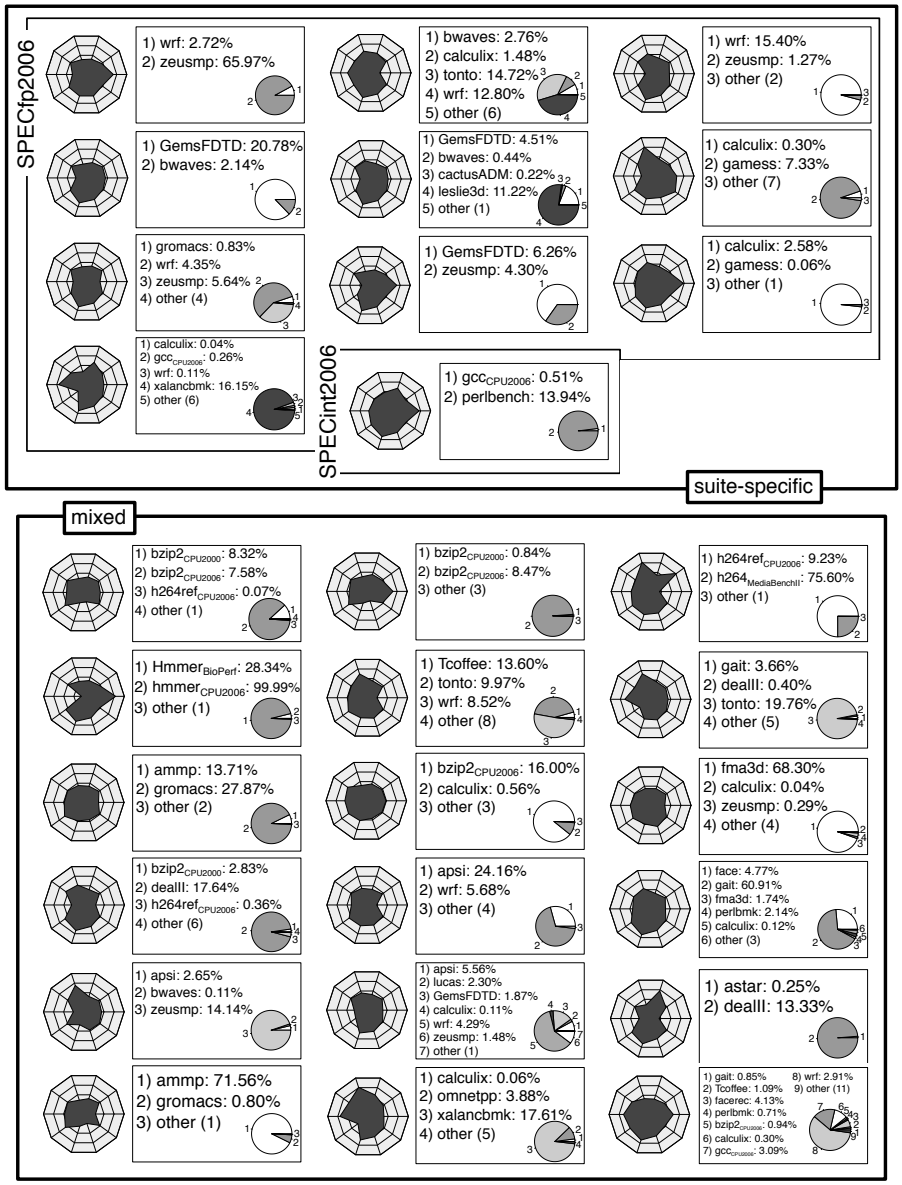


Figure 2.13: Kiviat plots (part III) representing the prominent phase behaviors.

of twolf is characterized by fairly average behavior except for low temporal locality, indicated by the low probability of LRU stack distance being smaller than 8 k, and the lack of very good per-instruction spatial locality for memory writes, indicated by the low probability of the stride distance being smaller than 64 (see Figure 2.11, bottom row). Another example is the kiviat diagram that represents the runtime behavior of povray (see Figure 2.12, third row from below). It is clear that povray shows more extreme behavior, as indicated by the kiviat diagram which shows a low amount of available ILP, a large instruction memory footprint and a (very) low spatial locality for local write operations.

Per-benchmark phase behavior These kiviat plots also present an interesting view on per-benchmark phase behavior. One interesting example is the runtime behavior of Fasta, which is split up into two major phases. The kiviat plots provide an easy-to-understand view on how these unique phase behaviors differ from each other: the two first diagrams in Figure 2.12 reveal three differences in terms of the key workload characteristics. The most striking difference is observed in the ratio of NOP instructions in both phase behaviors; the first phase shows a relatively low ratio, while in the second phase the ratio is fairly large, i.e., higher than the mean ratio plus one standard deviation. The two other differences are observed in terms of per-instruction spatial locality of memory write operations for small stride distances and the amount of available ILP. For finger, we identify three major unique phase behaviors (see Figure 2.11, top two rows). Various differences are again easily recognized between these phases, in terms of the instruction memory footprint, the amount of available ILP and the temporal locality.

Cross-benchmark phase behavior The suite-only phase behaviors reveal a limited overlap between several SPECfp2006 workloads and also between two SPECint2006 workloads, but only involves a very small fraction of the runtime behavior of one of the benchmarks in most cases.

The mixed phase behaviors are observed across different benchmarks from various benchmark suites; in general, these mixed clusters represent more average behavior than the benchmark-specific and suite-specific clusters, with some notable exceptions. One such exception is the phase behavior which is shared between the hmmer bench-

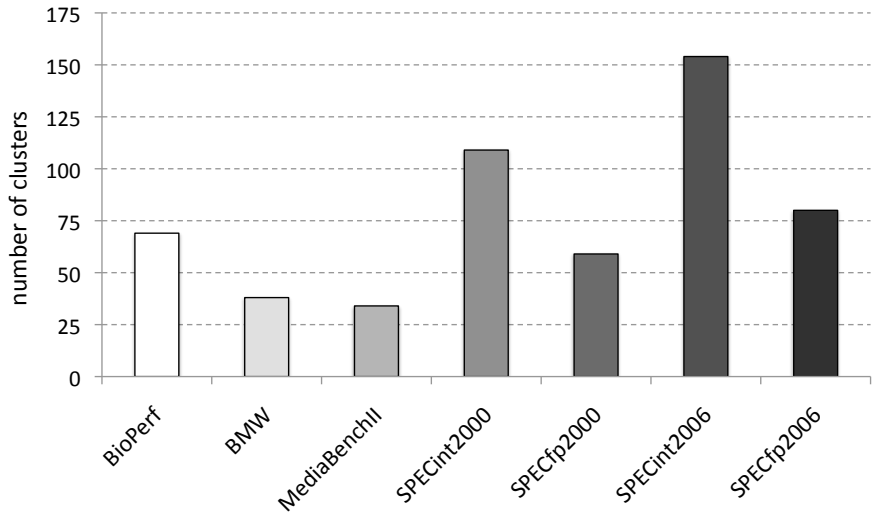


Figure 2.14: Workload space coverage per benchmark suite.

marks part of the BioPerf and SPEC CPU2006 benchmark suites, which shows a large amount of available ILP and a very small probability of the LRU stack distance being smaller than 2. Other examples are the phase behavior shared between the h264 and h264ref benchmarks, and the astar and dealII benchmarks. Some of the mixed phase behaviors also only involve just a very small fraction of one of the workloads involved. For applications that are part of multiple benchmark suites, such as bzip2, h264/h264ref and hmmer just a partial overlap of the phase behavior is observed, which suggests that a different version of the application or different types of input files significantly affect the runtime behavior.

2.3.2 Coverage, diversity and uniqueness of benchmark suites

We now look into the coverage, diversity and uniqueness of the various benchmark suites; for this purpose, we now consider all phase-level behaviors (see Section 2.2.3), not just the most prominent ones as in the previous section. Subsequently, we discuss the implications of these observations for simulation-based performance evaluation.

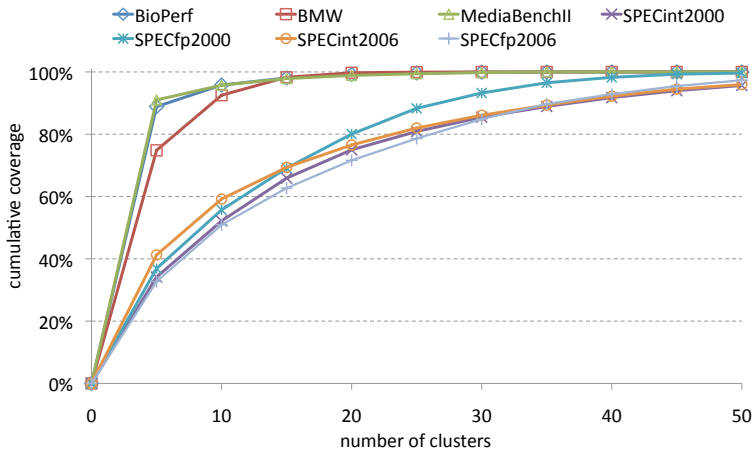


Figure 2.15: Cumulative coverage per benchmark suite as a function of the number of clusters.

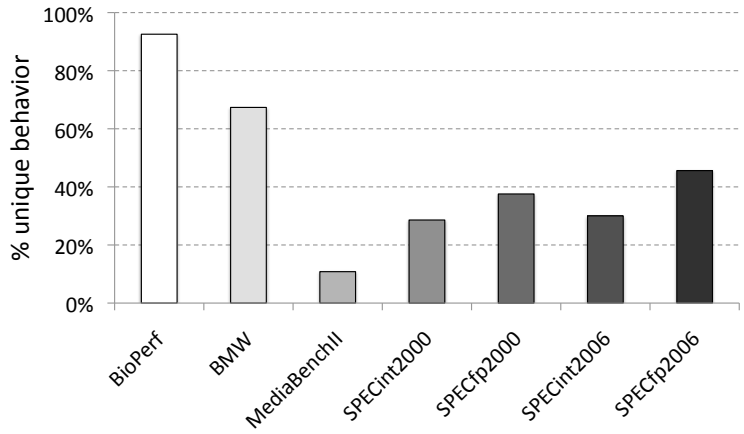


Figure 2.16: Fraction of a benchmark suite that represents unique program behavior not observed in the other benchmark suites.

Coverage of the workload space

First, we quantify a benchmark suite's workload space coverage, or how much a benchmark suite covers the entire workload space. Figure 2.14 shows the number of clusters (out of the 300) that represent some part of the benchmark suite, e.g., 154 and 80 of the clusters represent (at least one interval from) SPECint2006 and SPECfp2006, respectively.

We observe that the SPEC CPU2000 and CPU2006 integer benchmarks cover the largest part of the workload space, followed by floating-point benchmarks of the same suites. This reflects SPEC CPU's property of being a general-purpose benchmark suite. Note that the CPU2006 suites cover a broader part of the workload space than the CPU2000 suites. The emerging benchmark suites, BioPerf, BioMetricsWorkload and MediaBench II cover a narrower part of the workload space, i.e., they deliver significantly less phase behaviors, reflecting the fact that these benchmark suites are tied to a specific application domain.

Benchmark suite diversity

Next, we quantify the diversity within a benchmark suite. This is done by computing the cumulative number of clusters needed to represent a given fraction of the given benchmark suite. The results are shown in Figure 2.15: the cumulative coverage is shown per benchmark suite as a function of the number of clusters. For example, this graph shows that about 20 clusters are required to cover 72% of the SPECfp2006 benchmark suite; or, only 5 clusters are required to cover 88% of the BioPerf benchmark suite. The lower the curve for a given benchmark suite, the more clusters are required to cover a given percentage of the entire benchmark suite, and thus the higher the diversity. We observe that the domain-specific benchmark suites show a relatively low diversity compared to the general-purpose benchmark suites.

We thus conclude that BioPerf, BioMetricsWorkload and MediaBench II cover a much narrower part of the workload space than SPEC CPU, and in addition, the number of distinct behaviors within these benchmark suites is much smaller than for SPEC CPU. This analysis thus provides experimental evidence for the intuitive understanding that domain-specific benchmark suites represent a smaller part of the workload space than general-purpose benchmark suites do.

Uniqueness of each benchmark suite

We now quantify a benchmark suite's uniqueness with respect to the other benchmark suites. To do so, we compute the fraction of a given benchmark suite for which the relevant phase behaviors are specific to that particular benchmark suite, i.e., which do not represent any behavior of benchmarks from other benchmark suites (see Figure 2.16). For example, 93% of the BioPerf benchmark suite execution is represented

by either benchmark-specific or suite-specific clusters. In other words, 93% of the BioPerf benchmark suite exhibit unique program behavior not observed in other benchmark suites. This is the highest fraction observed among the benchmark suites analyzed. Also BioMetricsWorkload exhibits a fairly large fraction of unique program behavior with 67%, as opposed to MediaBench II, which only contains 11% of phase behavior not observed in other benchmark suites. The floating-point SPEC CPU benchmarks exhibit more unique behavior than the integer benchmarks, for both CPU2000 and CPU2006, but significantly less than the BioPerf and BioMetricsWorkload benchmarks.

Implications

The results obtained in the previous sections present a number of implications to performance evaluation. First, since SPEC CPU2006 exhibits a larger diversity than its predecessor (see Figure 2.15), this implies that a larger number of representative samples or simulation points need to be simulated for CPU2006 as for CPU2000 in order to cover all major phase-level behaviors in the benchmark suite. Second, both BioPerf and BioMetricsWorkload show a large fraction unique behavior not observed in the other benchmark suites. Therefore, including these benchmark suites into the experimental setup will yield additional insights. This is not the case for MediaBench II however; also considering this suite would only add to the simulation time required.

2.4 Related work

There exists a large body of work on workload characterization and its applications, which we summarize in this section.

Workload characterization in general

Next to traditional workload characterization studies, various authors have looked into the redundancy that is often present in benchmark suites, and use various subsetting techniques to counter this redundancy, e.g., to limit simulation time [100, 108]. Other work in workload characterization focuses on understanding how benchmarks evolve over time. One example is a study on how four subsequent generations of the SPEC CPU benchmark suite (CPU89, CPU92, CPU95 and

CPU2000) have evolved [66]. In that study, it is concluded that none of the inherent workload characteristics changes as dramatically as the dynamic instruction count. It was also observed that the temporal locality has become increasingly poor over time, while other characteristics have remained more or less the same. Another study looked into how benchmark drift affects processor design [109]. There, the authors conclude that benchmark drift can have a significant impact on the performance of next generation of microprocessors if the design of the next generation is driven solely by yesterday's benchmarks. This observation also motivates the work presented in this chapter, namely that an accurate workload characterization methodology is required for comparing emerging workloads against existing workloads, so that the microprocessors of the next generation perform well on future workloads.

Microarchitecture-independent workload characterization

Earlier work has recognized the strong correlation between the code that is executed and performance [6, 69, 70, 84, 93]. The SimPoint tool builds on this observation by using code signatures to select sampling units to be used during sampled simulation [93]. This is done by exploiting the fact that execution intervals that execute similar code stress the microprocessor in similar ways, i.e., they exhibit similar microarchitecture-dependent workload characteristics such as cache miss rates, branch misprediction rates, CPI, etc. Thus, code signatures allow for identifying phases in workloads, independently of the microarchitecture. However, because these code signatures are tied to the particular program being run, they can not be used for comparing the inherent workload behavior of different programs.

Therefore, researchers use workload characteristics instead for comparing programs. Previous work has characterized benchmarks at the programming level by counting the number of assignments, the number of if-then-else statements, the number of function calls, the number of loops, etc [102], or uses various workload characteristics at the Fortran programming language level such as operation mix, number of function calls, number of address computations, etc [89].

More recent work on studying program similarity uses statistical data analysis techniques on lower-level program characteristics. Some of these studies advocate the use of microarchitecture-independent workload characteristics solely [35, 86], as we have done in Sec-

tion 2.1.3. Others use a mixture of microarchitecture-dependent and microarchitecture-independent characteristics [36]. A completely different approach for identifying similarity across workloads is based on the Plackett-Burman design of experiments [107]. In this work, benchmarks are classified based on how they stress the same processor components to similar degrees.

Studying phase-level workload behavior

In recent literature, different approaches have been proposed to identify program phases. Some work identifies and predicts program phase behavior based on microarchitecture-dependent characteristics [33]. The advantage of microarchitecture-independent workload characterization, as we used in this chapter, is that it applies across different microarchitectures – this is especially valuable to exploit the program phase behavior to drive software or hardware optimization.

Several approaches to characterize program phase behavior independent of the microarchitecture have been proposed. One approach proposes to track the instruction footprint to detect transitions between program phases [32]. Another one is to compute Basic Block Vector (BBV) code signatures – a BBV computes the number of times a basic block has been executed in a given instruction interval. Next to showing that BBVs correlate strongly with performance metrics [69], others relate the phase behavior to methods and loops being executed [59]. Other studies identify phase behavior using other microarchitecture-independent workload characteristics, such as memory access patterns [70, 92] or a wide variety of workload characteristics [35]. Just like in the work we presented in this chapter, these approaches have the advantage over instruction footprints and BBVs that they can be used to compare the phase behavior of different workloads.

Various researchers have proposed to exploit phase behavior for a variety of applications. One application to phase analysis is hardware adaptation for energy saving [13, 32, 59, 95]. The idea there is to adapt the hardware on a per-phase basis so that energy consumption is reduced while not affecting overall performance. Another application is software profiling and optimization [44, 79]. Yet another application is simulation acceleration [35, 84, 93] by picking and simulating only one representative simulation point per phase, as done by the SimPoint tool.

2.5 Summary

Most workload characterization studies which look into the runtime program behavior of well-established or emerging workloads limit themselves to hardware performance counter-based and/or aggregate workload characterization. In this chapter, we illustrated that major pitfalls are associated with these approaches. Hardware performance counter metrics may hide the inherent program behavior. Aggregate workload characterization fails to capture the time-varying behavior of workloads.

We presented a phase-level workload characterization methodology [52], motivated by said pitfalls. We propose a set of microarchitecture-independent workload characteristics that capture the true inherent runtime behavior of programs, which allow for more informative workload characterization studies. Using these characteristics, we subsequently devised a methodology for feasible phase-level workload characterization. The key enablers in our proposal are different machine learning techniques such as Principal Component Analysis, cluster analysis (k-means clustering), and genetic algorithms. Together, these techniques allow for capturing the essential information that is present in a data set of phase-level microarchitecture-independent workload characteristics for a large collection of applications. Using our step-wise methodology, a set of key workload characteristics is identified which can be used to visually represent a limited set of phase behaviors which cover the largest part of the entire data set.

The proposed methodology was applied to a collection of microarchitecture-independent workload characteristics for execution intervals of 77 benchmarks taken from 5 different benchmark suites, resulting in a data set of 90 workload characteristics for over one million data points. The 100 most prominent phase behaviors, which collectively cover almost 90% of the entire workload space, were visually represented using kiviat diagrams showing just 10 key microarchitecture-independent workload characteristics, which capture the major differences in phase behavior. Studying these prominent phase behaviors in terms of coverage of the workload space, and diversity and uniqueness of each the benchmark suites confirmed the general-purpose nature of the SPEC CPU benchmark suites, and yields valuable insights on the domain-specific benchmark suites BioPerf, BioMetricsWorkload and MediaBench II.

Chapter 3

Analyzing Performance Trends

The purpose of computing is insight, not numbers.
Richard W. Hamming

Evaluating the performance of computer systems for a range of applications, whether it is done through simulation or through real hardware execution, is and will always be a key aspect of computer science and engineering. This is well recognized, and significant efforts have been made to make the process of performance evaluation more rigorous, for example by making standardized benchmark suites available that can be used by both academia and industry.

Well-established organizations such as SPEC[1], EEMBC[2] and TPC[3] have taken this one step further. They not only provide industry-standard benchmark suites for workloads, but they also collect performance numbers obtained with these benchmark suites on a wide range of computer systems and make these numbers publicly available. For example, SPEC provides performance numbers for benchmarks from various application domains such as compute-intensive workloads, Java workloads, graphics, web servers, mail servers, network file systems, etc.

Usually, the data that is made available consists of performance numbers for a number of individual benchmarks on a potentially large set of commercially available computer systems. Collectively, these performance numbers provide a wealth of information. For example,

[1] `http://www.spec.org`
[2] `http;//www.eembc.org`
[3] `http://www.tpc.org`

for the SPEC CPU2000 benchmark suite, performance numbers are available for 26 benchmarks for over 1,000 machines.

Given the large amount of data provided, it is often difficult to quickly gain insight from the wealth of performance data, i.e., it is hard to see how different systems compare against each other for different types of applications. Thorough performance analysis using a straightforward process of studying simple plots of the performance data available for tens of benchmarks and hundreds to thousands of computer systems in a spreadsheet is both insufficient and too time consuming. In this chapter, we present a methodology for the comprehensive analysis of a potentially large data set of performance numbers. The methodology presented relies on the statistical data analysis technique Principal Component Analysis, which is used in this chapter as a feature extraction technique.

3.1 Processor Performance Visualizer

In order to quickly and easily recognize performance trends from a database containing performance numbers for several benchmarks which were run on a large number of computer systems, we present the Processor Performance Visualizer (PPV) methodology. The key aspect of this methodology is the extraction of underlying dimensions in the data set using Principal Component Analysis, which we discuss in Section 3.1.1. Subsequently, the computer systems are represented in a low dimensional space constructed using selected principal components. Using an interactive tool which allows for easy navigation in a 3D-space representing the performance data allows for easily recognizing performance trends, as described in Section 3.1.2. Combining the observed trends with an interpretation of the principal components enables obtaining insight quickly and easily.

3.1.1 Finding performance trends using PCA

As described in Appendix A.1, Principal Component Analysis (PCA) can be viewed as a feature extraction technique, which extracts so-called principal components. These principal components are ordered according to the amount of variance in the original data set they explain. Therefore, one can easily capture the major trends in the data set using just a small number of principal components.

In the context of this chapter, the input to PCA are the performance numbers for the benchmarks on a set of computer systems. In other words, the variables are different benchmarks, and the instances are the computer systems for which performance is being evaluated. PCA will compute new variables, which represent uncorrelated underlying dimensions. More formally, PCA will transform a data set containing performance numbers for p benchmarks $S_1, S_2, \ldots, S_p$ on m computer systems into p principal components $Z_1, Z_2, \ldots, Z_p$. Each principal component Z_i is a linear combination of the input dimensions $S_j, 1 \leq j \leq p$, i.e., $Z_i = \sum_{j=1}^{p} a_{ij} S_j$, and captures a performance trend observed across the various computer systems.

An important concern when using PCA for feature extraction is the interpretability of the principal components. Even though a principal component is simply a linear combination of the dimensions of the original data set, attaching an intuitive meaning to each principal component can sometimes be surprisingly difficult, as was illustrated in Section 2.2.3 in the context of workload characterization.

As will become clear in the case studies presented in Sections 3.2 and 3.3, there are certain situations in which the interpretation of principal components is relatively easy. When each input dimension expresses the performance of one particular benchmark on different computer systems, it is often sufficient to identify one or more workload characteristics which discriminate between benchmarks with high factor loadings a_{ij} and those with low factor loadings. This allows for a concise interpretation of the principal components, which is then expressed as the average performance for a subset of applications with certain characteristics.

In the PPV methodology, each of the input dimensions is normalized to a zero mean and unit standard deviation, prior to applying PCA. This normalization step is required to put the benchmarks on a common scale. Also, the principal components that are included in the subsequent visualization step are normalized. Without this normalization step, a principal component that explains a large amount of variation would completely hide the patterns formed by principal components that explain less variation.

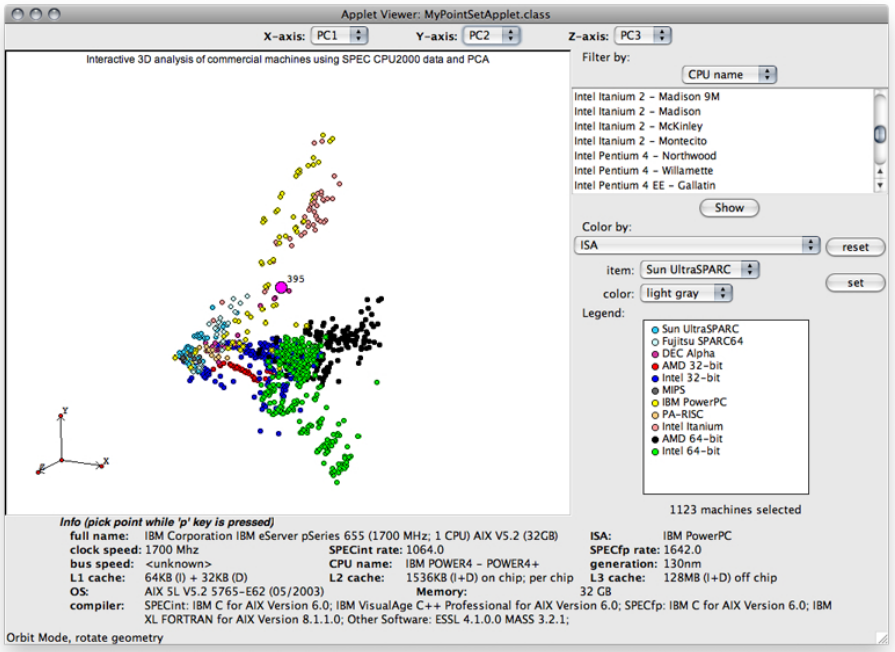


Figure 3.1: A screenshot of the Processor Performance Visualizer interactive user interface, which shows computer systems as dots in a 3-dimensional space spanned by principal components.

3.1.2 Interactive visualization

To further facilitate the discovery of interesting patterns and performance trends in the data set by looking at the data in terms of selected principal components, we developed an interactive PPV tool in the form of a Java applet. The input to the tool consists of the performance data for all available computer systems, expressed in terms of normalized principal components Z'_i. Next to this, the available meta-data for each of the computer systems is also fed into the tool. Examples include the system and components manufacturers, instruction set architecture (ISA), processor type and family, processor clock frequency, cache sizes, compiler used to build the benchmark applications, etc.

Using a straightforward user interface, the tool allows for visualizing the data as points in a 3D space, in terms of three selected principal components. The user can easily navigate through this performance space through various actions such as rotation, labeling data points by

color, filtering, zooming, selection of data points, etc. A screenshot of the user interface is shown in Figure 3.1.[4]

3.2 Case study: SPEC CPU2000

In a first case study which will illustrate the strengths of the Processor Performance Visualizer methodology described in Section 3.1, we analyze the base ratio performance numbers available for the SPEC CPU2000 benchmarks.

The data set of performance numbers for the SPEC CPU2000 benchmark was collected by the SPEC consortium during the lifetime of the benchmark suite – from the last quarter of 1999 to the first quarter of 2007 [5]. Computer system and processor manufacturers submitted performance reports for the CPU2000 benchmarks each time significant changes were made to their product line, yielding a wealthy source of performance data for the general-purpose microprocessors and computer systems produced during that time.

The SPEC CPU2000 benchmark suite consists of 26 benchmarks, split into two categories: 12 integer (SPECint) benchmarks and 14 floating-point (SPECfp) benchmarks. Performance data is available for computer systems with a large range of architectures, from server systems such as Intel Itanium and IBM POWER to desktop systems like Intel Pentium 4 and AMD Opteron. The microprocessor in these systems, which is the most important component for compute-intensive workloads, is implemented in a 500-nm to 65-nm CMOS technology and has a clock frequency ranging from 250 MHz to 3.8 GHz.

A performance number for one of the benchmarks on a particular computer system consists of the execution time speedup obtained relative to a reference system – a Sun Ultra5_10 workstation with a 300 MHz SPARC processor and 256 MB main memory. The entire data set comprises performance numbers for 1,381 and 1,399 computer systems, for SPECint and SPECfp, respectively. From this, we select the computer systems for which both base SPECint and SPECfp performance data were submitted on the same date – this results in a data set containing performance numbers for 1,123 computer systems. Table 3.1 gives an overview of the ISAs of these systems.

[4] The tool is available from `http://www.elis.ugent.be/~kehoste/PPV`.

[5] SPEC CPU2000 was retired in February 2007, and replaced by SPEC CPU2006.

Table 3.1: The computer systems considered in the SPEC CPU2000 case study, grouped by ISA.

architecture	# machines
Alpha	23
AMD x86 (32-bit)	28
AMD x86 (64-bit)	181
Intel x86 (32-bit)	250
Intel x86 (64-bit)	410
Intel Itanium (IA-64)	43
MIPS	10
PA-RISC	14
PowerPC	83
SPARC64	32
UltraSPARC	49

3.2.1 Interpretation of principal components

Collectively, the first three principal components explain 92.9% of the total variance observed in the data set; the first principal component explains 81%, the second principal component explains 7.7%, and the third principal component explains 4.2% of the total variance. We limit ourselves to three principal components because they explain most of the variance observed in the data set while enabling data visualization, i.e., the major performance trends will be preserved in the 3D space spanned by these underlying dimensions.

Figure 3.2 shows the factor loadings a_{ij} for the first three principal components. For example, the top graph in Figure 3.2 shows that the value along the first principal component for a given computer system is computed as $0.21 \times S_{n,ammp} + 0.18 \times S_{n,applu} + 0.20 \times S_{n,apsi} + 0.13 \times S_{n,art} + \ldots$, where $S_{n,b}$ indicates the normalized speedup for benchmark b.

Given the fact that the weights along the first principal component are approximately the same for all benchmarks, the first principal component thus represents the average normalized speedup across all benchmarks. This means that a computer system with a high value along the first principal component achieves relatively better average performance than a system with a low value along the first principal component. The only benchmark that has a relatively small weight in the first principal component is art; the reason is that the speedup

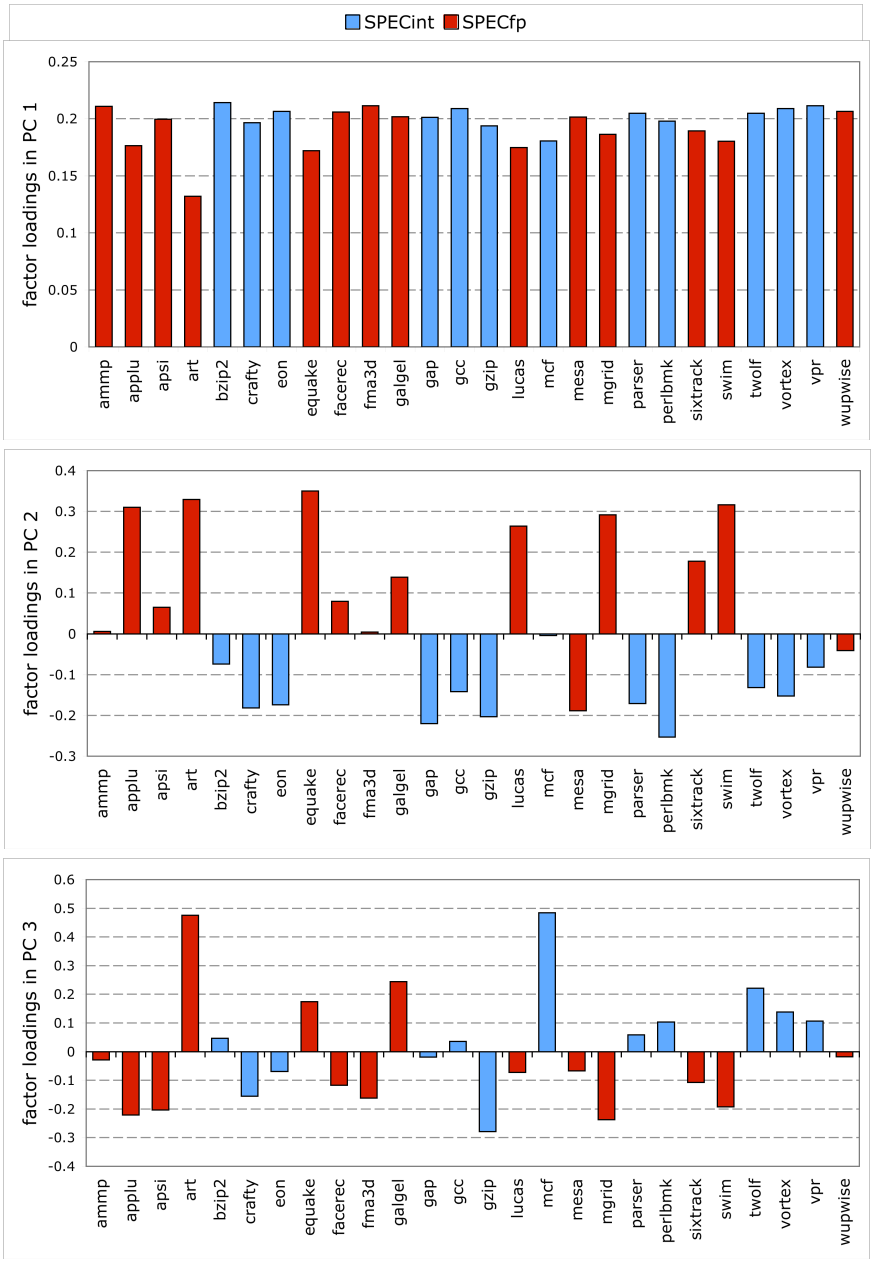


Figure 3.2: Factor loadings for the first three principal components, obtained by applying PCA to the SPEC CPU2000 data set of performance numbers for 26 benchmarks on 1,123 computer systems.

numbers vary widely for art across the various systems, more so than for any other benchmark. For about 10% of the machines, a speedup number greater than 8,236 (the maximum speedup number observed across the other benchmarks) is reported for art, with a maximum of 26,443; none of the other benchmarks show a speedup this large on any of the machines. The reason for this high speedup range lies in aggressive compiler optimizations [103] applied on some machines. This is recognized by PCA which assigns a lower factor loading to art in the first principal component, so that the performance numbers for art do not dominate the first principal component.

As expected, the most prominent dimension in the data set (the first and most significant principal component) correlates very well with average performance — the correlation coefficient between the first principal component and the SPECint and SPECfp scores is 0.969 and 0.974, respectively. This illustrates the huge performance increase achieved over the 7+ years during which the SPEC CPU2000 performance numbers were collected, through various enhancements in compiler support, architecture and chip technology.

The second principal component gives a positive weight to most of the floating-point benchmarks and a negative weight to all integer benchmarks. Hence, a machine with a high score along the second principal component yields relatively better normalized performance for the floating-point benchmarks than for the integer benchmarks. And vice versa, a low value along the second principal component indicates relatively better normalized integer performance, compared to floating-point performance. Two floating-point benchmarks, mesa and wupwise, get a negative weight, while all the other floating-point benchmarks are assigned a positive weight. This suggests that these benchmarks stress systems in a similar way as the integer benchmarks do. This observation is supported by the microarchitecture-independent workload characteristics. We found the ratio of floating-point operations and the branch misprediction rates for mesa and wupwise to be in the range of the integer benchmarks, and to be significantly different from the other floating-point benchmarks. More specifically, mesa and wupwise show fewer floating-point operations and higher branch misprediction rates than the other floating-point benchmarks, but similar to the integer benchmarks (see Figure 3.3).

In the third principal component, two benchmarks are assigned a significantly higher weight than the others, namely art and mcf. Both

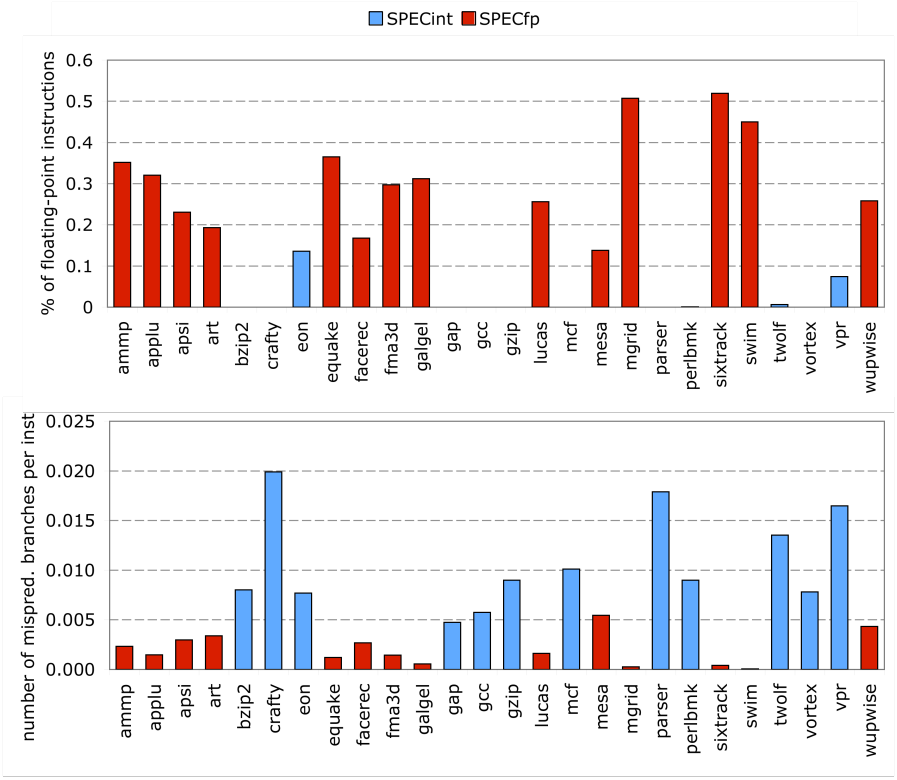


Figure 3.3: Percentage of floating-point instructions and number of mispredicted branches per instruction (GAg PPM predictor, 12 bit history) for the SPEC CPU2000 benchmarks. The mesa and wupwise benchmarks differ from the other SPECfp benchmarks, and lean more towards the values observed for the SPECint benchmarks.

benchmarks are known to stress the memory hierarchy subsystem more than any of the other SPEC CPU2000 benchmarks. Figure 3.4 illustrates this using microarchitecture-independent workload characteristics. Both art and mcf exhibit a low probability for the LRU stack distance to be smaller than 16 K. Taking into account that the LRU stack distances were measured using cache blocks of 64 bytes, these probabilities are a good indicator for the number of cache misses per instruction for a 1 MB sized L2 cache. Thus, computer systems with a high value along the third principal component yield relatively better performance for memory-intensive workloads with poor temporal data locality than systems with a lower value.

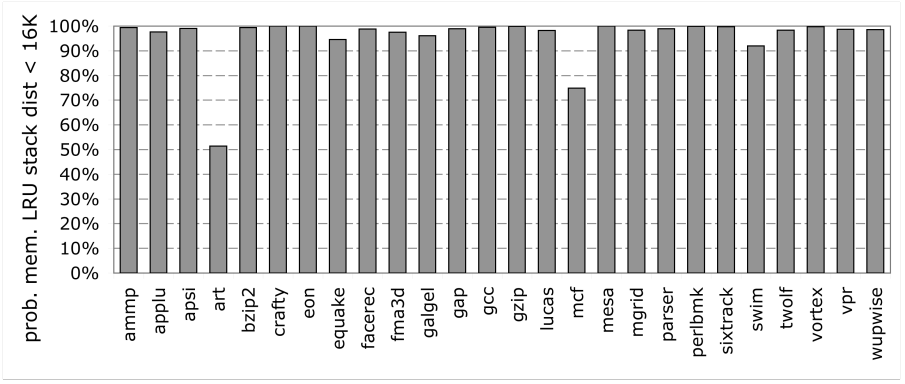


Figure 3.4: Probabilities of memory LRU stack distance being smaller than 16 K for the SPEC CPU2000 benchmarks. The low probabilities for art and mcf show that the memory accesses in these benchmarks exhibit poor temporal locality.

3.2.2 Discussion

Using the interpretation given to the first three principal components in the previous section, we now look into some interesting patterns and performance trends which can be observed. The interactive visualization tool described in Section 3.1.2 significantly facilitates finding these patterns and trends by browsing through the 3D-space spanned by the principal components. For the purpose of clarity however, we will rely on traditional 2D scatter plots to highlight the observations made.

Major performance trends

Visualizing the various computer systems in the 3-dimensional space spanned by principal components (PCs) yields the graphs shown in Figure 3.5. The 2-dimensional scatter plots show the second PC versus the first PC, and the third PC versus the first PC. Each dot in these graphs represents one system; there are 1,123 systems plotted in each graph; and the various colors encode various architectures. Many interesting insights can be gained from analyzing Figure 3.5 — we just give a couple of examples here for the sake of illustration.

The right bottom corner in the PC2 vs. PC1 plot in Figure 3.5 shows that the Intel x86 64-bit machines yield better average performance for the integer benchmarks than the other machines. The PowerPC and

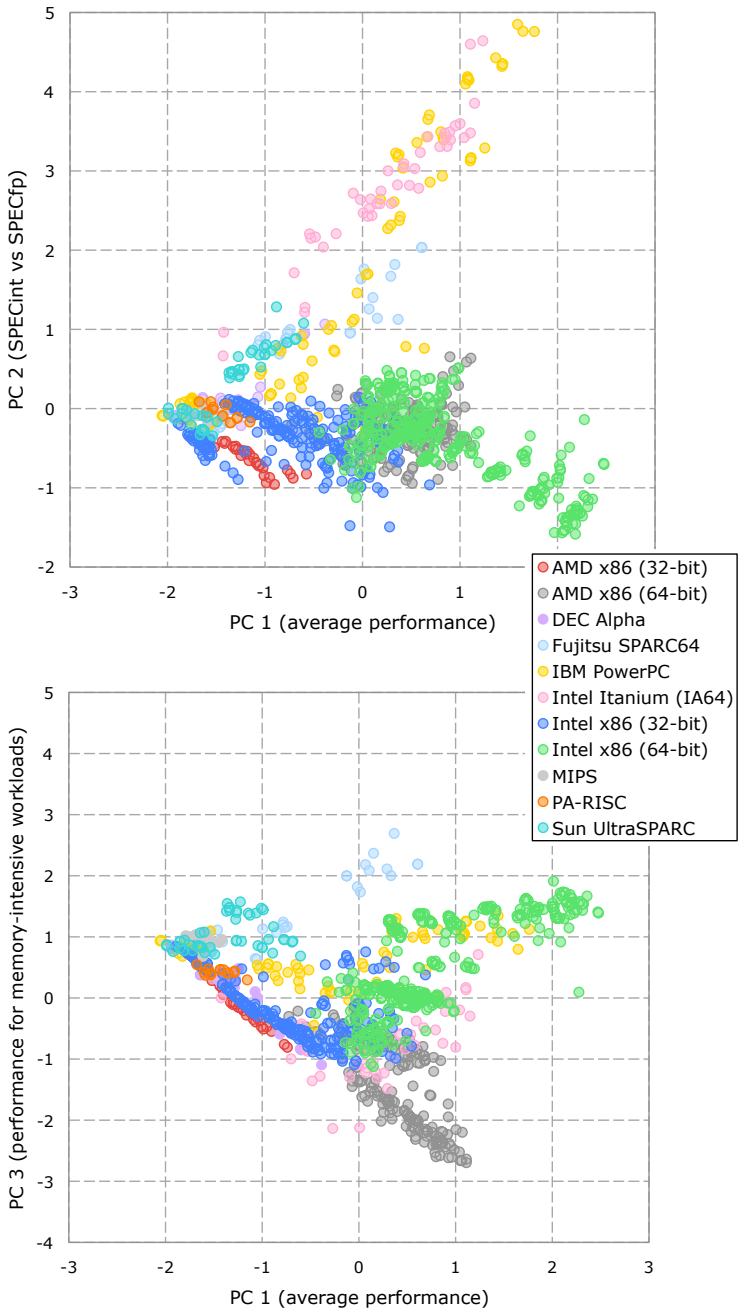


Figure 3.5: Visualizing the SPEC CPU2000 performance numbers in terms of the first three principal components. The colors represent different architectures.

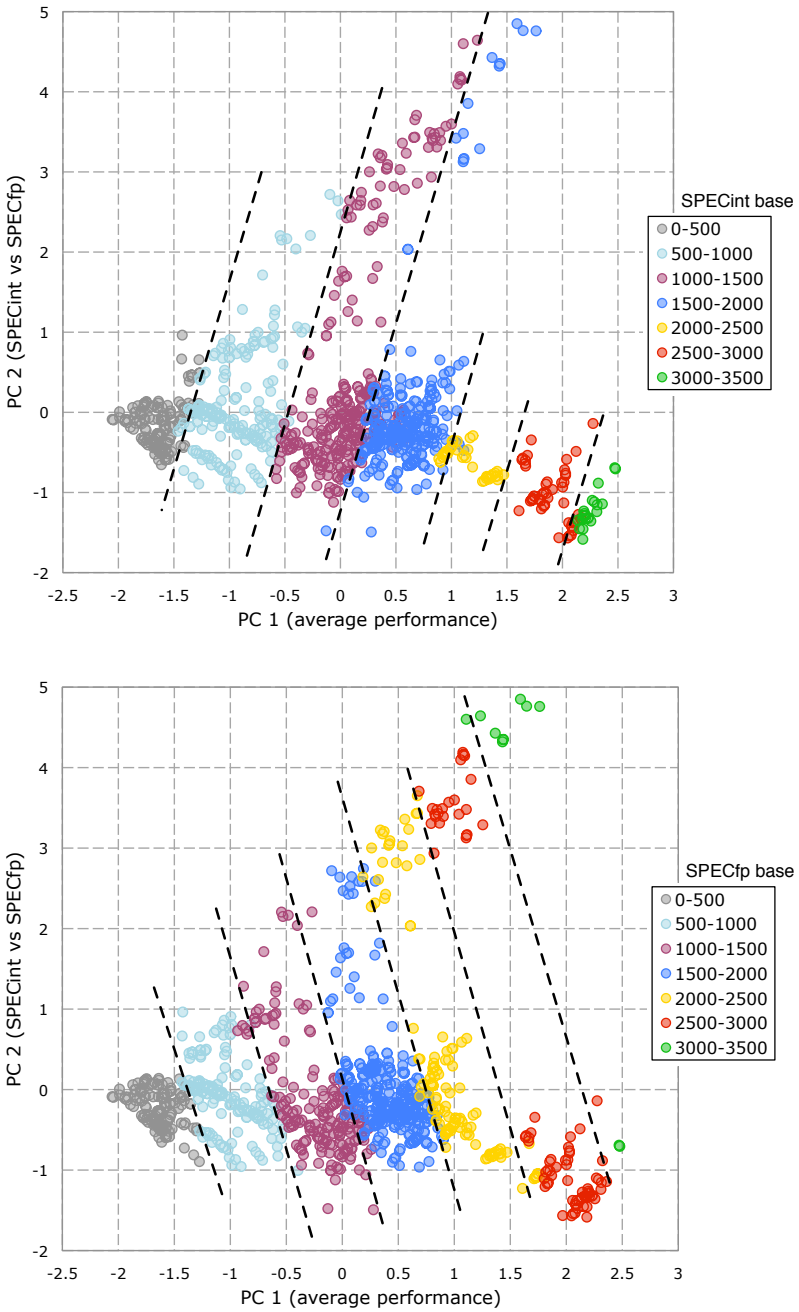


Figure 3.6: Visualizing PC2 vs. PC1 obtained from the SPEC CPU2000 performance numbers. The colors reflect ranges of average SPEC numbers: the top and bottom graphs use colors showing average CINT2000 and CFP2000 speedup numbers, respectively.

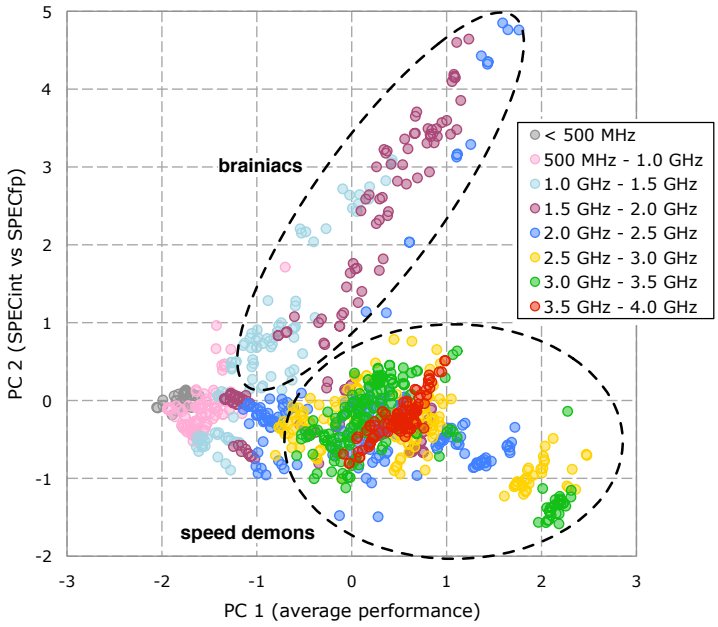


Figure 3.7: Visualizing the SPEC CPU2000 performance numbers in terms of the first three principal components. The colors reflect ranges of processor clock frequencies.

Itanium machines perform better on the floating-point benchmarks, see the upper part in the PC2 vs. PC1 plot. This is also illustrated in Figure 3.6 where the various computer systems are shown in terms of PC1 and PC2 categorized by average SPEC performance number; the top and the bottom graphs show the average SPEC performance numbers for the integer and floating-point benchmarks, respectively. The interesting observation here is that the performance wave of increased performance for the integer benchmarks has a different orientation than the performance wave for the floating-point benchmarks. This confirms the interpretation given to the second principal component as discriminator between systems performing better on integer or floating-point workloads. Systems in the lower half of the PC1-vs-PC2 graph obtain better SPECint performance than the systems with roughly the same value along PC1, i.e., comparable overall average performance, in the upper half of the graph. Likewise, systems in the upper half of the graph show higher SPECfp performance compared to overall equally performing systems in the lower half.

Another interesting observation is that the PC3 vs. PC1 plot in Figure 3.5 shows that although both the PowerPC and Itanium machines achieve similar average performance for the integer and floating-point benchmarks — values are obtained in the same range along the first two principal components — they seem to exhibit very different behavior in terms of the third principal component. This is due to the PowerPC machines performing relatively better than the Itanium machines for the memory-intensive benchmarks with poor temporal data locality (i.e., art and mcf). The same is observed when comparing Intel 64-bit versus AMD 64-bit systems: systems with an Intel processor perform better than AMD-based systems for memory-intensive workloads.

Figure 3.7 represents the same data in another way, by coloring the systems by their processor clock frequency. This clearly shows two groups of machines, namely the speed demons and the brainiacs. The speed demons achieve high performance mainly through high clock frequencies; the brainiacs on the other hand achieve high performance through a high instruction throughput per cycle (IPC) at a relatively low clock frequency. The speed demons appear to be the Intel and AMD 64-bit machines; these machines achieve high performance for the integer benchmarks through their high 2.5+ GHz clock frequencies. The brainiacs are the PowerPC and Itanium machines which achieve high performance on the floating-point benchmarks by achieving high instruction throughput per cycle at moderate clock frequencies of less than 2.5 GHz.

Intel NetBurst generation

We now look into systems with an Intel processor implementing the NetBurst architecture, see Figure 3.8. The various computer systems included are categorized by processor type. This plot clearly shows the evolution across the various generations of Intel NetBurst-based processors.

There are a number of interesting observations to be made here. First, it confirms the general expectation, namely average performance improves across Intel NetBurst processor generations, i.e., the data points go from left to right across the different generations. Second, the different processors within a single generation show a negative slope in both the PC2 vs. PC1 and PC3 vs. PC1 graphs. This suggests that increasing the clock frequency within a processor generation (which is the main contributor to the variation in performance within a proces-

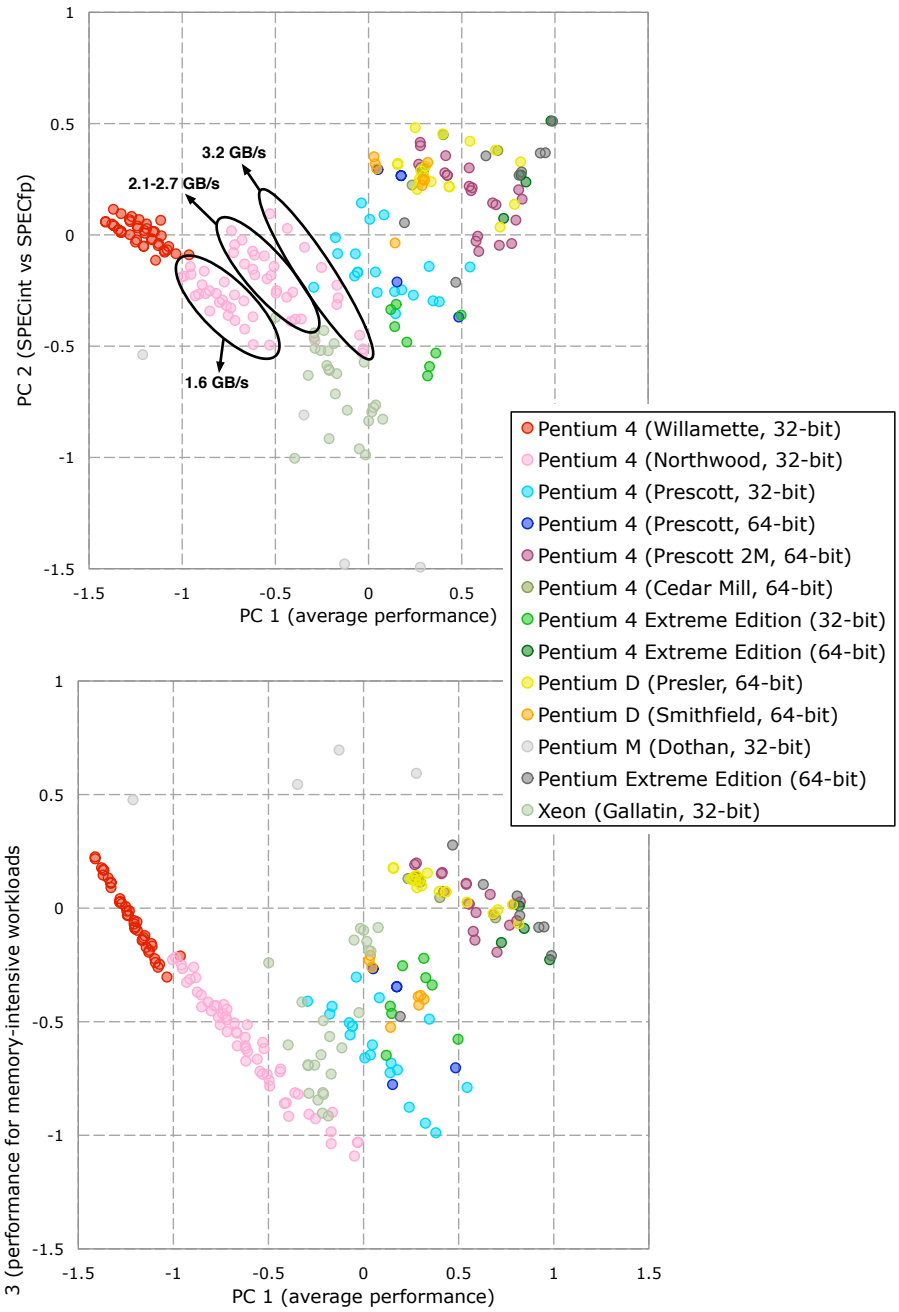


Figure 3.8: Detailed study of the processors that implement the Intel NetBurst architecture in terms of the second versus first principal components (top graph) and third versus first principal components (bottom graph).

sor generation) improves performance more for the compute-intensive and integer benchmarks than for the memory-intensive and floating-point benchmarks. The reason is that memory-intensive benchmarks spend more time waiting for memory — increasing clock frequency does not improve performance as much for memory-intensive applications as it does for compute-intensive applications. A third interesting observation is that three sub-generations can be observed within the Northwood processor generation, see Figure 3.8 (top graph). The three sub-generations represent machines with different memory bandwidth characteristics ranging from 1.6 GB/s, to 2.1-2.7 GB/s, up to 3.2 GB/s.

3.3 Case study: SPEC CPU2006

In our second case study, we apply the methodology presented in Section 3.1 to SPEC CPU2006, the latest SPEC benchmark suite for general-purpose workloads. For a detailed overview of the SPEC CPU2006 benchmark suite, we refer to Appendix B.5.

The performance numbers available for the SPEC CPU2006 benchmarks cover computer systems tested from the 3rd quarter of 2006 to the last quarter of 2009. Again, this represents a sufficiently wide range of systems to allow for identifying major performance trends.

The SPEC CPU2006 benchmark suite contains 29 benchmarks, split up in SPECint (12) and SPECfp benchmarks (17). Our data set contains the base ratio performance numbers for 1,040 computer systems, which are again relative to a reference system – mostly identical to the one used for SPEC CPU2000, but with 2 GB of main memory.

Table 3.2 gives an overview of the architectures of these systems. The vast majority consists of systems with an x86-64 processor, i.e., Intel NetBurst/Nehalem/Core or AMD Athlon/Opteron/Phenom/Turion. Thus, this case study will mainly focus on this group of systems, since most performance trends will only concern these systems.

3.3.1 Interpretation of principal components

Figure 3.9 shows the factor loadings for the first three principal components. Individually, each of these PCs explain 77.7%, 10.7% and 3.1% of the total variance, respectively, and collectively they explain 91.5% of the total variance. We will concentrate on just these three principal

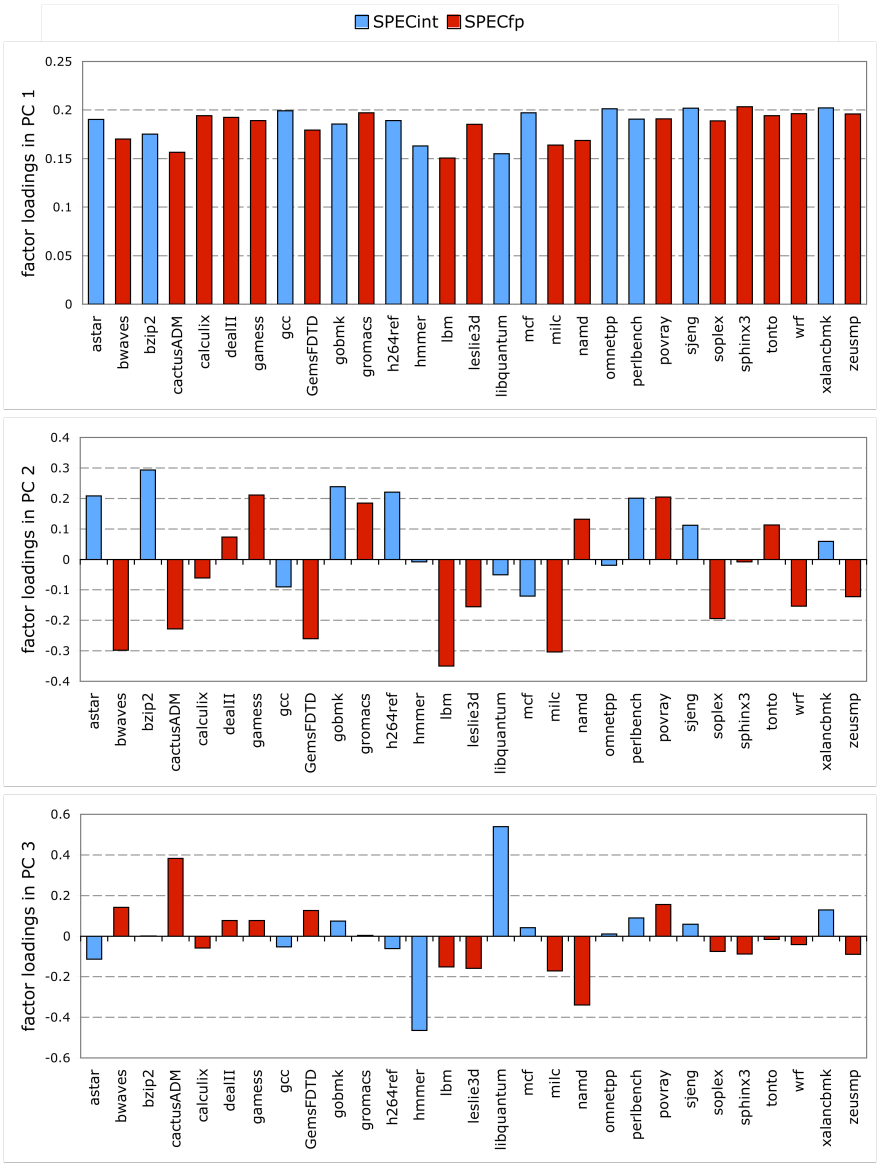


Figure 3.9: Factor loadings for the first three principal components, obtained by applying PCA to the SPEC CPU2006 data set of performance numbers for 29 benchmarks on 1,040 computer systems.

Table 3.2: The computer systems considered in the SPEC CPU2006 case study, grouped by ISA.

architecture	# machines
Intel NetBurst	18
Intel Core	524
Intel Nehalem	259
AMD Athlon 64	7
AMD Opteron K8	46
AMD Opteron K10	128
AMD Phenom	13
AMD Turion	3
Intel Itanium	19
IBM POWER	10
SPARC64	6
UltraSPARC	7

components, which are sufficient to bring forward interesting performance trends.

The weights along the first principal component are again approximately the same for all benchmarks, and thus also in this case study the most prominent underlying dimension represents average normalized speedup across all benchmarks.

The second principal component clearly discriminates between two groups of benchmarks. Several benchmarks, mostly SPECint workloads, obtain positive weights. Other benchmarks, clearly biased towards SPECfp workloads, obtain a negative weight. Figure 3.10 shows that most of the benchmarks that obtain a negative weight show relatively poor temporal locality: 2% or more of the memory accesses of these benchmarks correspond with an LRU stack distance over more than 128k, except for cactusADM. This is a strong indicator for a significant amount of long-latency memory accesses. Some benchmarks also showing poor temporal locality are not given a negative weight in PC2; the memory-intensity of their program behavior is hidden by either complex control flow behavior, causing a significant amount of branch mispredictions (e.g., omnetpp), or very good spatial locality (e.g., libquantum).

Thus, PC2 captures the differences in system performance for memory-intensive workloads with poor temporal locality. A computer

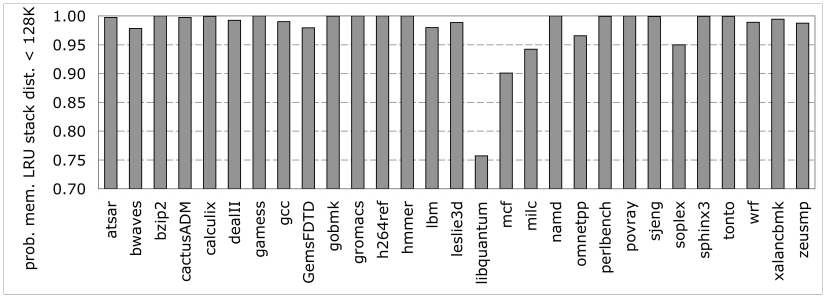


Figure 3.10: Temporal data locality quantified by the probability of the memory LRU stack distance being smaller than 128k.

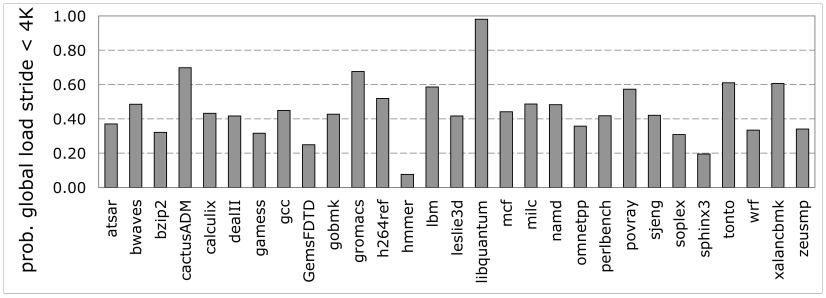


Figure 3.11: Spatial data locality quantified by the probability of the global load stride distance being smaller than 4096. A high weight in the third principal component can be linked to good spatial data locality, a low weight is observed for benchmarks with poor spatial locality.

system that obtains a high value along the second principal component performs better for workloads with good temporal locality, while systems that show a low value along PC2 yield better performance for memory-intensive workloads with a poor temporal locality.

For the third principal component, we observe high Pearson correlation coefficients of the group of microarchitecture-independent workload characteristics quantifying spatial locality with the factor loadings of PC3. Figure 3.11 shows that libquantum exhibits very good spatial locality: the addresses of over 98% of the memory read operations are within a 4096 boundary of the address of the previous memory read operation. For hmmer however, we observe that only 8% of all memory read operations access an address even remotely close to the previously touched address. This is not only a possible cause of a higher cache miss rate, but could also cause major paging activity, both of which are

likely to have a significant effect on performance.

The weights for the third principal component shown in Figure 3.9 match this discrepancy in spatial data locality: libquantum obtains a high positive weight, while hmmer is given a significant negative weight. Two other benchmarks that also jump out in terms of weights in the third principal component, i.e., cactusADM and namd, show no exceptional spatial data locality in Figure 3.11. However, other factors such as aggressive prefetching can again affect this significantly both in beneficial and destructive ways, and may explain the significant weights for these benchmarks. This analysis suggests that a computer system with a high value along the third principal component performs better for workloads with good spatial locality than systems with a similar average performance that show a lower value. Likewise, systems that show a low value for PC3 show better performance for workloads with irregular data access patterns than a performance-wise competitive system with a higher value along the third principal component.

3.3.2 Discussion

Visualizing the computer systems in terms of the first three principal components again allows us to identify interesting performance trends, see Figure 3.12.

The first most striking observation is the significant shift made by the Intel Nehalem generation of microprocessors. While the previous generations of Intel processors, i.e., those implementing the NetBurst or Core microarchitecture, are mainly yielding better performance for workloads with good temporal locality, this is apparently not the case for the Nehalem architecture. Figure 3.12 shows a clear shift towards lower values along the second principal component for the Nehalem systems, suggesting that significant improvements in the microarchitecture relieve part of the performance bottleneck for workloads with poor temporal locality in the NetBurst and Core microarchitecture. Both the Intel Itanium and IBM POWER systems show lower values along PC2 compared to (most of) the AMD Opteron and Intel Core-based systems, indicating that poor temporal locality is less of a bottleneck in these systems.

Another interesting observation is the direction in which the systems move in terms of the second principal component as their aver-

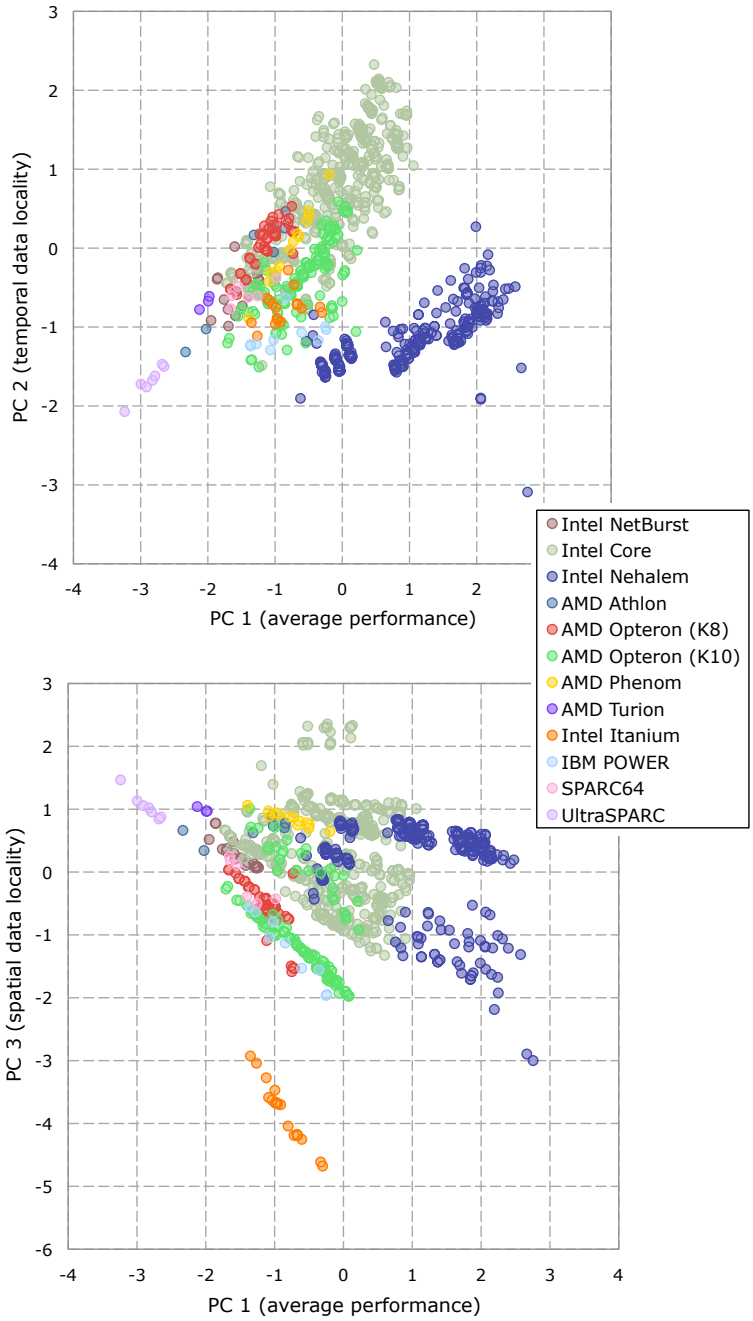


Figure 3.12: Visualizing the SPEC CPU2006 performance numbers in terms of the first three principal components. The colors represent different processor architectures.

age performance, captured by the first principal component, improves. As mentioned before, the largest contributor in improved performance within a single processor generation is the increased clock frequency. The up-right slope indicates that increasing the clock frequency mainly yields improved performance for workloads with good temporal locality. Indeed, when a particular workload is causing the processor to stall due to long-latency memory instructions, increasing the clock frequency will yield little performance improvement because it does not resolve the main performance bottleneck. Note that this is similar to the performance trend observed for the Intel NetBurst generation of microprocessors in Section 3.2.

The bottom graph in Figure 3.12 also reveals some interesting performance trends. Different waves of computer systems are observed. First, the Intel Itanium systems clearly separate themselves from the others in terms of performance for workloads with low spatial locality, e.g., hmmer. This can be easily explained by the specialized cache hierarchy, i.e., an L2 data cache of 256 kB and an on-chip L3 cache of 6 MB up to 12 MB. While the IBM POWER systems also provide large caches, up to 4 MB of L2 cache and 32 MB of off-chip L3 cache, the emphasis on workloads with a low spatial locality is less outspoken. The aggressive software prefetching performed by the compiler used to build workloads on Itanium systems, which results in an efficient use of the large caches, is presumably one of the reasons for this. Another observation is the two clearly separated groups of Intel Nehalem computer systems. The systems in the upper right part of the bottom graph all contain a microprocessor of a particular implementation of the Intel Nehalem generation, i.e., Intel Xeon Gainestown. The other group consists of systems with other processor implementations of the Intel Nehalem family. A likely cause of the separation in two distinct groups is the introduction of the QuickPath Interconnect (QPI), which replaces the front side bus (FSB).

Figure 3.13 again shows two differently oriented performance waves when the computer systems are colored by SPECint and SPECfp rates. The upper graph shows that computer systems that perform better for workloads with good temporal locality obtain higher SPECint rates than systems that perform better, relatively, for workloads with poor temporal locality. The bottom graph reveals the opposite trend in terms of SPECfp rate: systems that perform well relatively speaking for programs that suffer from a significant amount of long-latency memory accesses obtain higher SPECfp rates than the other systems.

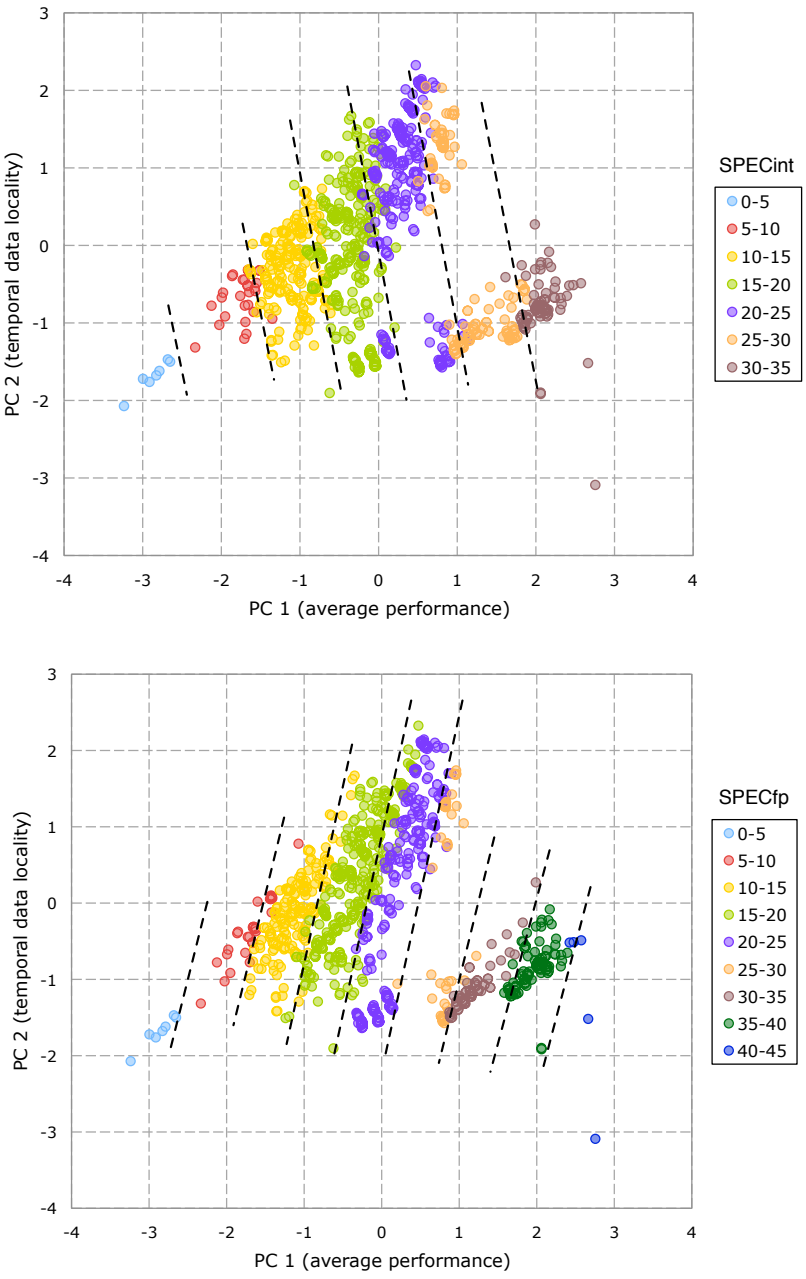


Figure 3.13: Visualizing the SPEC CPU2006 performance numbers in terms of the first two principal components. The colors reflect ranges of average SPECint and SPECfp base rate numbers.

3.4 Related work

Although interest into using statistics in microprocessor performance analysis has grown recently, this is the first work providing a comprehensive methodology for analyzing performance trends across a large data set of processor performance numbers.

Lilja [73] describes various statistical data analysis techniques that can be readily used by computer architects for taking meaningful conclusions from large data sets. These techniques include the t-test, linear regression modeling, design of experiments, etc. The Plackett and Burman design of experiments, which is a fractional factorial design of experiments, allows for identifying microarchitecture design parameters that have a large impact on overall performance using a limited number of simulations [107].

Other uses of statistical techniques are related to the non-determinism of computer systems which needs to be taken into account when evaluating performance. It was observed that the performance of a computer system with multiple single-core processors running a multi-threaded workload is susceptible to non-deterministic effects, i.e., subtle changes may lead to different interleavings and interactions between threads executing on such a system. In order to account for these non-deterministic effects, a methodology was proposed that introduces non-determinism in deterministic simulators and then uses statistics for computing confidence bounds [2].

The use of confidence bounds to determine how many sampling units to take for estimating uniprocessor performance was proposed by Conte et al. [27]. Others showed that a matched-pair comparison reduces the number of samples that need to be taken in order to estimate a change in performance between alternative processor architectures [38]. In other work, a statistical approach to selecting multiple starting points for simulating multi-program workloads on multi-thread processor architectures was used [99].

3.5 Summary

The wealth of performance numbers provided by benchmarking consortiums or corporations complicates understanding general performance trends across commercial machines. In this chapter, we proposed a performance analysis methodology and framework based on

Principal Component Analysis (PCA), a statistical data analysis technique which we use as a machine learning technique for feature extraction [56]. PCA reduces a large data set into a manageable data set which facilitates the ease of understanding. Including the set of microarchitecture-independent workload characteristics in the analysis allows for interpreting each of the principal components in terms of these workload characteristics, enabling valuable insights.

Applying PCA to the published SPEC CPU2000 and SPEC CPU2006 benchmark suite performance numbers yields various interesting insights in a limited number of plots that summarize the major performance trends. This analysis shows that the proposed methodology is a powerful tool in the toolbox of a performance analyst.

Chapter 4

Estimating Relative Computer System Performance

Prediction is difficult, especially about the future.
Niels Bohr

In benchmarking, the key challenge is to determine the computer system that yields the best performance for a given application-of-interest. Ideally, a user's application-of-interest is his/her best benchmark. However, in many practical circumstances the user has to rely on the performance scores of a standardized benchmark suite for estimating the performance of the application-of-interest for a number of reasons. First, it is too difficult or costly to port the application-of-interest to a wide range of platforms. Another reason is that there are many different systems for which the performance needs to be measured before making a choice about which computer system yields the best performance for the given application. Of course, purchasing all the systems that are of potential interest just to evaluate their performance is infeasible.

A popular tool for estimating the performance of software applications on an unavailable platform is detailed cycle-accurate simulation. However, next to not solving the porting problem, simulation is very time consuming and thus ill-suited in practice to tackle this key challenge in benchmarking.

This motivates us to come up with a different solution to this ubiquitous problem in benchmarking, which is presented in this chapter. By means of machine learning techniques including a genetic algorithm and k-nearest-neighbors (kNN), we outperform current practice,

which relies on average system performance, in ranking a set of computer systems in terms of their respective performance for a particular application-of-interest.

4.1 Performance estimation framework

The framework we present relies on the performance numbers available for standardized benchmark suites on the computer systems of our interest. As a part of our framework we collect microarchitecture-independent workload characteristics for the application-of-interest, capturing and quantifying its inherent runtime behavior, and relate it to the benchmarks in the standardized benchmark suite. We then use the knowledge of similarity between the inherent program behavior of the application-of-interest and the corresponding benchmarks to estimate the performance for the application-of-interest. In other words, we use the standardized benchmarks as proxies for our application-of-interest.

4.1.1 Relating differences in inherent workload behavior to performance differences

The key issue in a performance estimation framework that uses program similarity based on microarchitecture-independent workload characteristics is to determine how differences in terms of these workload characteristics translate into differences in performance. For this, we use a genetic algorithm that learns how to rescale the workload space spanned by the workload characteristics so that the Euclidean distance in the workload space becomes a good measure for performance differences.

Figure 4.1 illustrates the framework that we propose for estimating the performance based on the similarity in terms of inherent program behavior. The framework assumes the availability of a benchmark suite. For each of the benchmarks, we collect the microarchitecture-independent workload characteristics as well as performance numbers for various computer systems. The performance numbers could be obtained from simulation or from real hardware execution.

Both the workload characteristics and the performance numbers are then used by a genetic algorithm[1] to gradually evolve a set of weights

[1] See Appendix A.2 for a detailed discussion on genetic algorithms.

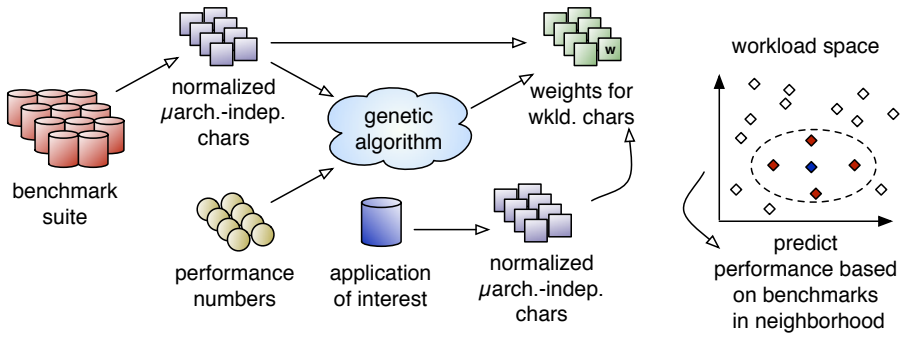


Figure 4.1: The framework proposed in this chapter for estimating performance for an application-of-interest based on microarchitecture-independent program characteristics.

for each of the workload characteristics. Weighting the workload characteristics is done to tweak the Euclidean distance measure used to determine the benchmark proxies. This is motivated by the intuitive notion that different workload characteristics have a different impact on overall performance than others; e.g., the amount of ILP typically has a much larger impact on performance than the fraction of integer instructions.

The genetic algorithm will favor sets of weights, each of which is represented by a vector of floating-point values, that yield a better match between the Euclidean distances obtained for each pair of benchmarks in terms of the weighted workload characteristics and the corresponding performance differences. This is done through a leave-one-out cross-validation mechanism, in which each of the benchmarks is in turn considered to be the application-of-interest, and its performance is estimated based on the performance for the other benchmarks. The fitness score used by the genetic algorithm is the average relative estimation error across the set of benchmarks, i.e.,

$$f = \frac{1}{n_b} \sum_b \frac{p_b - p'_b}{p_b}$$

in which p_b represents the real performance number for a benchmark b, p'_b is the estimated performance number for that benchmark, and n_b is the number of benchmarks in the training set.

4.1.2 Estimating performance for a particular application

After this training step to obtain weights for each of the workload characteristics, we can map the application-of-interest for which we want to estimate performance in the workload space by weighting its workload characteristics. This way, we are able to determine the most similarly behaving benchmarks for the application-of-interest, allowing for estimating the performance of this application.

The actual performance estimation is done through the k-nearest-neighbors technique, which only takes into account the performance of the closest benchmarks in the neighborhood of the application-of-interest. A benchmark that is in the neighborhood of the application-of-interest is further referred to as a *proxy*. We estimate the performance of the application-of-interest as the weighted average of the performance numbers of the proxies. Weighting for each proxy is done based on the Euclidean distance between the proxy and the application-of-interest in terms of the weighted workload characteristics. The weight w_p for a particular proxy p is inversely proportional to the distance d_p, and is computed as

$$w_p = \frac{1}{d_p \cdot \sum_{i=1}^{k} \frac{1}{d_i}}$$

with k the number of retained proxies for the application-of-interest. Note that the sum of all weights w_i (i in $[1, k]$), is ensured to be one by normalizing each weight with respect to the sum of all proxy weights.

We focus on estimating speedups relative to a base system, rather than estimating raw performance numbers. Estimating relative performance differences often suffices in practice, for example to rank different computer systems. The estimated speedup s of the application-of-interest is computed as the weighted harmonic average over the speedups of the proxies:

$$s = \frac{1}{\sum_{p=1}^{k} \frac{w_p}{s_p}}$$

4.1.3 Discussion

An inherent limitation of this performance estimation framework is that accurate performance estimation is difficult for an application-of-interest that is isolated in the workload space. This would indicate

that the application-of-interest is dissimilar to all of the programs in the benchmark suite in terms of its inherent workload behavior. Hence, it is to be expected that an accurate performance estimation will be difficult to obtain based on the almost non-existing similarity of the application-of-interest with the programs in the benchmark suite.

As a result, an important issue to our performance estimation framework is which programs to select for inclusion in the benchmark suite, i.e., the benchmark suite should be diverse enough to cover a wide range of program behaviors. In the experimental evaluation section, we use SPEC CPU as our benchmark suite because the SPEC website records performance numbers for all of the SPEC CPU2000 and CPU2006 benchmarks for a large variety of platforms.

Although we can reasonably expect that for some of the benchmarks it will be hard to estimate performance based on the other benchmarks, showing that our performance estimation framework works well using a standardized benchmark suite has a lot of practical appeal. People can compare their application-of-interest versus the SPEC CPU benchmarks based on inherent program behavior and make performance estimations using the publicly available SPEC CPU results for a large number of commercial machines.

4.2 Experimental evaluation

4.2.1 Experimental setup

We evaluate our performance estimation framework using SPEC published speedup ratios that cover various commercial machines with different ISAs, compiler settings and microprocessors. In particular, we use the SPEC CPU2000 and CPU2006 benchmark suites in different evaluation experiments, each with two different performance data sets: one for a small hand-picked subset of systems which differ significantly in terms of microprocessors, and another one containing all available performance numbers on over 1,000 systems each. Tables 4.1 and 4.2 detail on the data sets used for both the SPEC CPU2000 and CPU2006 benchmark suites, respectively. The performance numbers used are the speedup ratios with base optimization reported on the SPEC website[2]. Note that we limit ourselves to aggregate performance metrics here; SPEC only provides aggregate performance numbers per benchmark.

[2]See `http://www.spec.org/cpu2000` and `http://www.spec.org/cpu2006`.

Table 4.1: Performance data sets used for the SPEC CPU2000 benchmark suite; the left column presents details for the small hand-picked subset of significantly different systems, whereas the right column details on the full data set of available performance numbers.

	small subset	*full data set*
Alpha	2	23
AMD x86 (32-bit)	2	28
AMD x86 (64-bit)	6	181
Intel x86 (32-bit)	7	250
Intel x86 (64-bit)	8	410
Intel Itanium (IA-64)	3	43
MIPS	1	10
PA-RISC	-	14
PowerPC	2	83
SPARC64	2	32
UltraSPARC	3	49
total number of systems	36	1,123

The evaluation is performed through leave-one-out cross-validation, in which each benchmark is regarded as the application-of-interest in turn. The remaining benchmarks then form the benchmark suite as referred to in Section 4.1.1 on which the training of the weights for the workload characteristics is performed. The microarchitecture-independent workload characteristics used to capture the inherent workload behavior are the ones discussed in Section 2.1.3. Before computing the Euclidean distance in terms of these workload characteristics or weighting them, we normalize each characteristic to a zero mean and unit variance, to ensure that they are all put on a common scale.

The genetic algorithm is configured to evolve a single population of 100 entities in the form of vectors of floating-point values, each representing a set of weights for the workload characteristics. Between each generation, the best 20 sets of weights in terms of the fitness score, i.e., the average estimation error, are used to construct the next generation. Except for the initial generation, of all entities in a generation, 85% are constructed by means of the crossover mixing operator, with a crossover rate of 0.25; the remaining 15% of the entities are obtained through multi-point drift mutation, with a mutation rate of 0.25. Convergence is assumed when no more significant changes in the fitness score are observed for three subsequent generations.

Table 4.2: Performance data sets used for the SPEC CPU2006 benchmark suite; the left column presents details for the small hand-picked subset of significantly different systems, whereas the right column details on the full data set of available performance numbers.

	small subset	*full data set*
AMD x86 (Athlon 64)	-	7
AMD x86 (Opteron K8)	1	46
AMD x86 (Opteron K10)	5	128
AMD x86 (Phenom)	2	13
AMD x86 (Turion)	-	3
Intel x86 (NetBurst)	2	18
Intel x86 (Core)	15	524
Intel x86 (Nehalem)	5	259
Intel Itanium (IA-64)	8	19
IBM POWER	9	10
SPARC64	3	6
UltraSPARC	-	7
total number of systems	50	1,040

4.2.2 Evaluation

Estimating computer system ranking

To evaluate the framework on the different performance data sets, we rank the computer systems based on estimated performance. We use the Spearman rank correlation coefficient to assess the quality of an estimated ranking compared to the real ranking, which yields a value between -1 and 1; the former indicates a reversed ranking, the latter indicates a perfect ranking. We compare the Spearman rank correlation coefficients obtained through our framework with those obtained using current practice, i.e., the coefficients for the estimated rankings based on average computer system performance.

We evaluate the accuracy of our framework by treating the benchmarks as the application-of-interest one by one. The parameter k representing the number of proxies used by the k-nearest-neighbors technique is varied, to assess the effect of this parameter on ranking quality. Where detailed results regarding the number of proxies are omitted, we limit ourselves to discussing the results for a fixed value of k showing the best average Spearman rank correlation across all benchmarks.

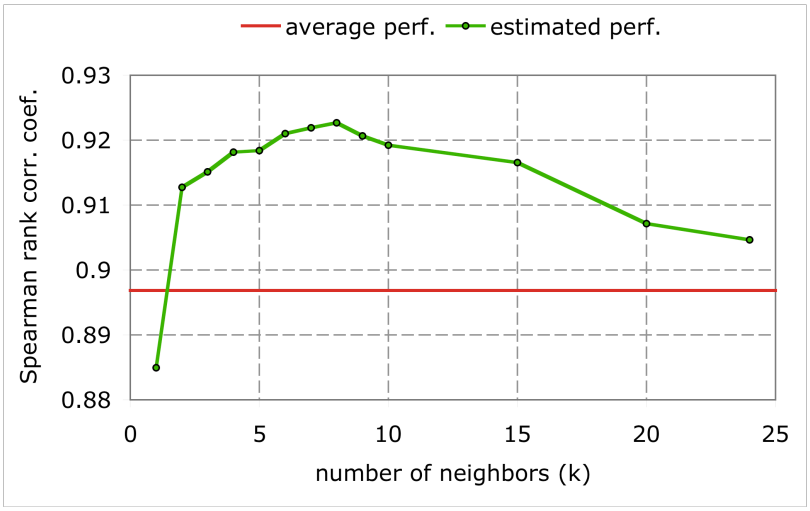


Figure 4.2: Average Spearman rank correlation coefficients obtained for the SPEC CPU2000 benchmark suite, the data set of performance numbers for 1,123 computer systems and different values for the number of proxies k.

Estimating rankings for the SPEC CPU2000 benchmarks

Figure 4.2 shows the average Spearman rank correlation coefficients obtained for the SPEC CPU2000 benchmark suite and the data set containing all 1,123 computer systems. The green curve shows the average correlation coefficients using our estimation framework for different values of the number of proxies k, the red line indicates the correlation coefficient obtained by ranking the computer systems based on average performance. These results clearly illustrate the importance of the number of neighbors used to estimate performance; retaining not enough neighbors results may result in rankings of disappointing quality, while retaining too many neighbors could also result in a significant degradation of the Spearman correlation coefficient. For this particular data set the best result is obtained using 8 proxies, resulting in a Spearman rank correlation coefficient of 0.923. This is a significant improvement over the quality of rankings obtained based on average performance, yielding a rank correlation of 0.897.

The per-benchmark correlation coefficients obtained both using our framework and current practice based on average performance for the large SPEC CPU2000 data set are shown in Figure 4.3. For 22 out of the 26 benchmarks, the ranking obtained based on the performance estima-

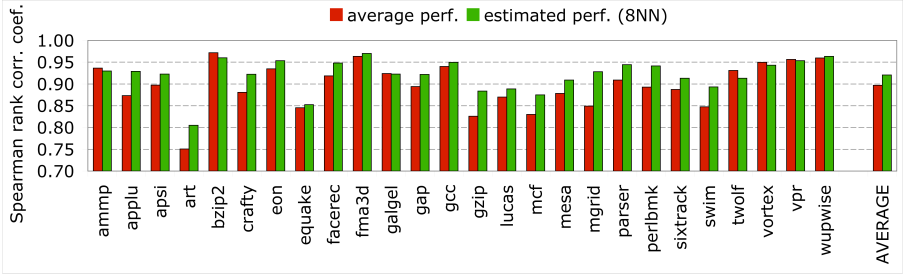


Figure 4.3: Per-benchmark Spearman rank correlation coefficients obtained for the SPEC CPU2000 benchmarks and the data set of performance numbers for 1,123 computer systems, both using average performance and our performance estimation framework, using 8 nearest neighbors.

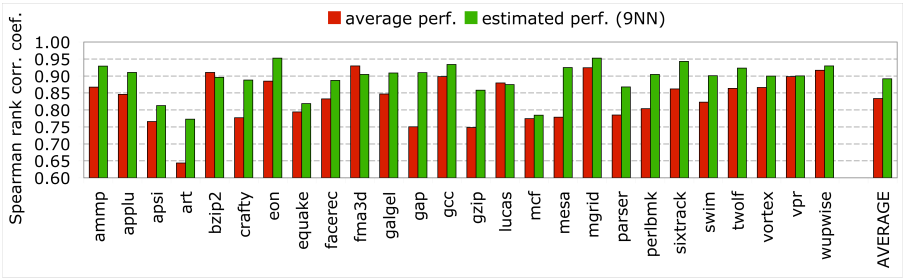


Figure 4.4: Per-benchmark Spearman rank correlation coefficients obtained for the SPEC CPU2000 benchmarks and the hand-picked subset of performance numbers for 36 computer systems, both using average performance and our performance estimation framework, using 9 nearest neighbors.

tions of our framework is better than the one obtained based on average performance. For several benchmarks that show relatively low correlation coefficients using current practice, e.g., art, gzip, mcf and swim, we observe significant improvements using our estimation framework. The fairly low correlation coefficients obtained for art are not unexpected; the art benchmark is a significant outlier in terms of memory access behavior and retains excessive speedups on a large number of systems due to aggressive compiler optimizations [103], making it a significant outlier with respect to the other benchmarks.

For the small data set containing performance numbers for 36 hand-picked computer systems, current practice results in a Spearman correlation coefficient of 0.833. Through our estimation framework, the best result is obtained for 9 nearest neighbors, corresponding with a corre-

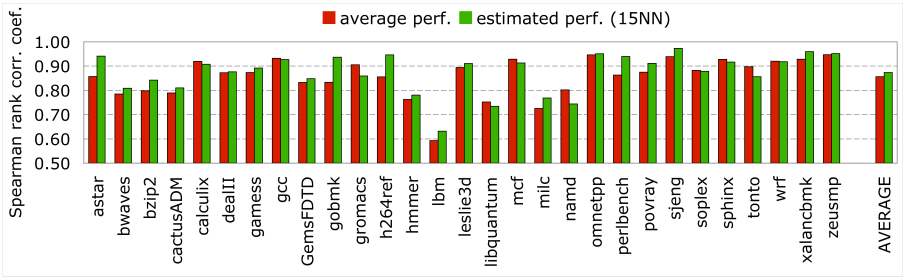


Figure 4.5: Per-benchmark Spearman rank correlation coefficients obtained for the SPEC CPU2006 benchmarks and the data set of performance numbers for 1,040 computer systems, both using average performance and our performance estimation framework, using 15 nearest neighbors.

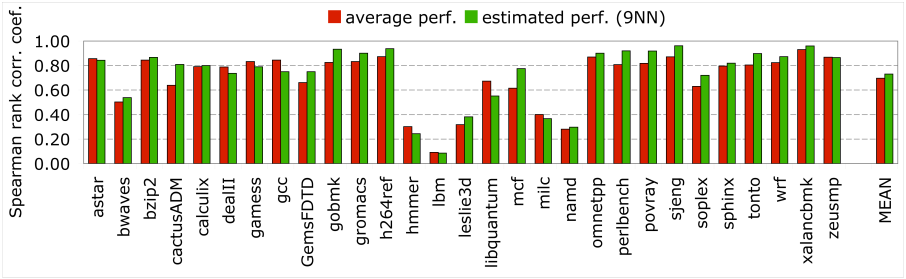


Figure 4.6: Per-benchmark Spearman rank correlation coefficients obtained for the SPEC CPU2006 benchmarks and the hand-picked subset of performance numbers for 50 computer systems, both using average performance and our performance estimation framework, using 9 nearest neighbors.

lation coefficient of 0.892. The detailed results are shown in Figure 4.4, again showing large improvements for various benchmarks, and just modest drops for some benchmarks with high Spearman correlation coefficients. For this data set, the rankings obtained using our performance estimation framework outperforms current practice for 23 out of the 26 benchmarks.

Estimating rankings for the SPEC CPU2006 benchmarks

Figures 4.5 and 4.6 show the per-benchmark and average correlation coefficients for the SPEC CPU2006 benchmarks, using the full data set of performance numbers for 1,040 systems and the small hand-picked data set of 50 systems, respectively.

For the full data set, our framework yields an average Spearman correlation coefficient of 0.874, improving on a coefficient of 0.857 obtained based on average performance; we outperform current practice for 19 out of 29 benchmarks. Likewise, for the small data set our framework shows a significant improvement over current practice: we obtain an average Spearman correlation coefficient of 0.731 while ranking based on average performance yields a value of 0.696, and outperforms current practice for 20 out of the 29 benchmarks.

The low average Spearman correlation coefficients for the small data set are easily explained by the various outlier benchmarks resulting in very low quality rankings, i.e., hmmer, lbm, leslie3d, milc, namd, etc. This corresponds to the observations made in Chapter 2 about the SPEC CPU2006 benchmark suite covering a broader part of the entire workload space, and retaining a large amount of benchmark-specific inherent program behavior.

Using other machine learning techniques

Next to k-nearest-neighbors combined with a genetic algorithm for weighting the workload characteristics we also tried a variety of other machine learning techniques, including decision trees, linear regression, regression trees, support vector machines (SVMs), etc. Neither of these techniques yielded results even close to the ones we obtained with the framework we presented.

There are a number of reasons for this. First, there is the limited amount of training data available. Since both the SPEC CPU2000 and CPU2006 benchmark suites only contain a small number of benchmarks, i.e., 26 and 29, respectively, we only have a limited amount of performance data available. Lots of machine learning techniques tend to overfit on small data sets, i.e., yield a performance model that produces accurate estimations for the data points in the training set but fail to generalize to others. Another difference between k-nearest-neighbors and the other techniques we evaluated is that the latter uses the actual training data when estimating for a new data point, while the other techniques construct some kind of performance model during training and never look back at the original data when estimating for a new data point. Both these differences contribute to significantly better results for our framework compared to other machine learning techniques.

4.3 Related work

The fundamental facilitator for our performance estimation approach is a good quantitative measure for program similarity. Several researchers have proposed methods for quantifying program similarity. Saavedra and Smith [89] use the squared Euclidean distance computed in a benchmark space built up using dynamic program characteristics at the Fortran programming language level such as operation mix, number of function calls, number of address computations, etc. Yi et al. [107] use a Plackett-Burman design for classifying benchmarks based on how the benchmarks stress the same processor components to similar degrees.

Several researchers have proposed benchmark suite composition techniques [35, 37, 86], which first measure a number of program characteristics, then apply PCA, and finally apply cluster analysis in order to find distinct groups of program behavior. A representative is then chosen from each cluster for inclusion in the benchmark suite. The key idea is to select benchmarks so that all major program behaviors are represented in the benchmark suite. This technique can be used for building a benchmark suite that covers the benchmark space well, or it could be used to build a reduced benchmark suite from an existing benchmark suite. This reduced benchmark suite yields accurate performance predictions compared to the original benchmark suite.

A preliminary study by Phansalkar and John [85] used this workload characterization methodology consisting of PCA and cluster analysis to predict performance for individual benchmarks. An important issue with PCA however is that the distance measure in the benchmark space may not relate well to the performance differences across various platforms [58]. This motivates the use of weights for each of the workload characteristics in our performance estimation framework.

Another approach to the benchmarking problem that we address in this chapter is analytical modeling. Ideally, an analytical model would consume microarchitecture-independent characteristics as well as microarchitecture parameters and produce accurate performance estimates of the given application on the given microarchitecture. The work that gets close to such an approach is the superscalar processor model presented by Eyerman et al. [39, 67] that estimates performance based on microarchitecture-dependent characteristics such as cache miss rates and branch misprediction rates. Various researchers have

proposed techniques to predict cache miss rates based on microarchitecture-independent characteristics such as the stack distance, see for example [111]. However, we are unaware of any work that proposes a superscalar processor model based on microarchitecture-independent characteristics solely — the major impediment for achieving this is a good model for estimating branch misprediction rates based on microarchitecture-independent characteristics.

4.4 Summary

In this chapter, we proposed an approach for addressing the ubiquitous problem in benchmarking which is ranking a set of computer systems for a given application-of-interest. The key idea is to compare inherent workload characteristics of the application-of-interest against the same characteristics for all programs in the standardized benchmark suite. Based on the inherent similarity of the application-of-interest with the benchmarks in the benchmark suite, a number of proxies are identified and a performance estimation can be made using the performance numbers of the proxies [58].

The estimation framework was evaluated using the SPEC CPU2000 and CPU2006 benchmark suites and two performance data sets, a small one with a hand-picked selection of computer systems which vary significantly in terms of microprocessors, and another one containing all available data for over 1,000 systems. Our framework yields Spearman rank correlation coefficients for the estimated ranking of systems that are significantly higher than the ones obtained through ranking by average system performance. For the CPU2000 data sets, we obtain correlation coefficients of 0.923 and 0.892 using our estimation framework, for the large and small data sets, respectively; based on average system performance, correlation coefficients of 0.897 and 0.833 are obtained for the same data sets. With the CPU2006 benchmarks, we obtain similar results: correlation coefficients of 0.874 and 0.731 using our estimation framework versus 0.857 and 0.696 based on average system performance, for the large and small data sets, respectively. Because of the interpolating nature of the k-nearest-neighbors technique, applications-of-interest that show inherent program behavior that differs significantly from that of all the benchmarks result in relatively low Spearman rank correlation coefficients.

Chapter 5

Constructing Compiler Optimization Levels

There are no solutions... there are only trade-offs.
Thomas Sowell

Modern compilers are complex pieces of software. They consist of various front-ends for different programming languages, a middle-end which performs aggressive optimization and a variety of back-ends which take care of the actual code generation for the different instruction-set architectures supported. Adjusting any part of the compiler, either for maintenance or to add additional functionality, requires deep knowledge of the existing code base. This is a non-trivial task because of the inherent complexity of compiler software. For example, the GNU Compiler Collection (GCC)[1] consist of well over 2 million lines of code, with over half a million lines in the core compiler [81].

The complexity is even more apparent for the parts of the compiler that perform various optimizations which target a variety of objectives, including performance, compilation cost and code size. In a modern compiler, a broad collection of optimizations is available, often providing over 100 different optimizations. Anticipating the efficacy of these optimizations for one particular objective is not trivial though. The effect of a compiler optimization is highly dependent on the code being compiled, and the hardware platform for which the code is being compiled. In addition, multiple compiler optimizations interact in complex and in many cases even counter-intuitive ways, or at least in ways that are hard to reason about. To make things even worse, different objec-

[1] `http://gcc.gnu.org/`

tives may be affected conflictingly by compiler optimizations. Put together, this constitutes a challenging task for compiler developers who wish to provide an easy-to-use optimizing compiler framework to their end users.

This is a well recognized problem, and to facilitate the end user in determining an appropriate selection of compiler optimizations, compiler developers typically provide a set of standard optimization levels, such as `-O1`, `-O2`, `-O3` and `-Os`. Each of these optimization levels combine various compiler optimizations and provide different trade-offs in terms of code quality, compilation time and code size. For users willing to trade additional compilation time for better code quality, the highest level of optimization (e.g., `-O3`) is worth a try. If on the other hand compilation time is a concern, more so than code quality, `-O1` may be the optimization level of choice. Finally, if code size is of primary importance, for example for embedded applications, then `-Os` is a suitable optimization level.

The construction of these optimization levels however is troublesome. The search space is huge given the large number of available compiler optimizations. For example, in the 4.2.4 version of the GCC compiler which we use throughout this chapter, 68 different optimizations are used in the various optimization levels [2] (`-O1`, `-O2`, `-O3`, `-Os`). This results in a huge space with 2^{68} (in the order of $\approx 10^{20}$) possible optimization levels. An exhaustive search in such a huge space is obviously infeasible, because of the time required to compile and run the benchmarks for each of the optimization levels considered. Even if we would be able to evaluate one optimization level every second, it would take over 9 trillion years to get through the entire search space.

Therefore, compiler developers heavily rely on their experience, intuition and various heuristics in their definition of optimization levels. These heuristics typically look like: optimizations that do not increase compilation time and are likely to produce good code, should be activated at `-O1`; optimizations that tend to increase code size but will likely result in better code quality, should be activated at `-O2` and disabled at `-Os`; and optimizations that typically require a lot of compilation time, and might lead to even better code, should be activated at `-O3`.[3] Ishizaki et al. [62] describe such a manual optimization level

[2] We only consider the optimizations that can be turned on and off using available command-line flags; various optimization levels also perform optimizations that are not controllable by external flags.

[3] See `http://gcc.gnu.org/onlinedocs/gcc-4.2.4/gcc/Optimize-Options.html`

selection process for a dynamic Just-In-Time compiler. However, not only is relying on such heuristics suboptimal and does it require deep knowledge of the various available optimizations and their possible interactions, it is also very time-consuming and labor-intensive.

In this chapter, we propose Compiler Optimization Level Exploration (COLE), a framework to automatically construct optimization levels that represent optimal trade-offs between multiple objective functions, such as performance, compilation time, code size, etc. COLE not only relieves the compiler developer from the tedious and time-consuming task of manually building optimization levels, it also (most likely) yields better performing optimization levels. To the best of our knowledge, we are the first to propose automated multi-objective compiler optimization level exploration, of which the COLE framework is a prototype example. The core of COLE consists of a multi-objective evolutionary search algorithm, a machine learning technique shown to be an efficient way of searching in huge compiler optimization spaces.

5.1 Compiler Optimization Level Exploration

Before describing the COLE framework itself, we first explain the notion of Pareto optimality in the context of compiler optimization levels.

5.1.1 Pareto optimality

In order to explain what Pareto optimality means, we need to introduce some terminology. A given compiler optimization level is called *Pareto dominant* with respect to another optimization level if the given optimization level achieves a better score for at least one objective function while achieving the same (or a better) score along the other objective functions. A *Pareto optimal* compiler optimization level L is an optimization level for which there exist no other optimization levels that are Pareto dominant with respect to L. Multiple Pareto optimal compiler optimization levels can co-exist to form a so called *Pareto frontier* or *Pareto set*. In other words, the Pareto set collects all the Pareto optimal compiler optimization levels. Once the Pareto frontier is identified, the compiler developer or end user can then select a compiler optimization level that trades off the various objective functions according to his or her needs.

Figure 5.1 shows an example Pareto frontier when two objectives

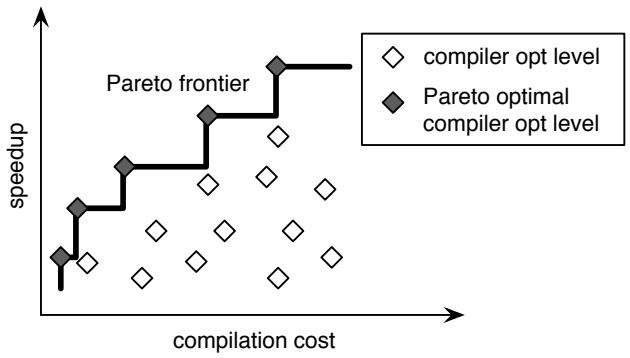


Figure 5.1: An example Pareto frontier in a multi-objective design space, using speedup (higher is better) and compilation cost (lower is better) as objectives.

are considered, compilation cost and speedup; an optimization level is represented by a single dot. Both compilation cost and speedup are relative to an optimization level which performs little or no optimization (e.g., GCC's `-O0`). In terms of compilation cost, an optimization level is better than another level if it results in a smaller compilation time. On the other hand, if a certain optimization level results in a lower execution time, thus yields a higher performance than another level, it is considered to be better in terms of speedup.

5.1.2 Multi-objective exploration

COLE employs multi-objective evolutionary searching for identifying Pareto optimal optimization levels — the search algorithm used by COLE is based on SPEA2 [113], which is an improved version of the well-established Strength Pareto Evolutionary Algorithm (SPEA) [114].

In this section, we limit ourselves to the aspects of the particular search algorithm used by COLE which are important for the following discussion and evaluation. For more details on the SPEA2 algorithm, and its relation with genetic algorithms, see A.2.5.

In the context of this chapter, each optimization level being considered during the exploration is modeled by an entity, which represents a (candidate) solution in the search space. For this work, we only consider enabling or disabling optimizations, along with an additional parameter representing the aggressiveness in which the optimizations are performed (`O1`, `O2`, `O3`, `Os`). Thus, each entity consists of a boolean

vector plus one extra discrete-valued parameter. Combining two entities to form two offspring entities is done using the crossover mixing operator, while randomly tweaking a single entity is done using the multi-point drift mutation operator. Convergence is assumed when no change is observed in the Pareto frontier for a number of subsequent generations.

We do not tweak value parameters that steer the optimizations, nor do we change the order in which optimizations are performed. This does not limit the applicability of our framework: including additional non-boolean parameters or taking the order in which optimizations are being performed into account is a matter of changing the entity definition and adjusting the crossover and mutation operators accordingly.

Throughout the entire chapter, we will focus on two metrics as optimization objectives. The first metric is the compilation cost incurred by an optimization level relative to that of the standard optimization level `-O0`, which performs little or no optimization. The second objective is the execution time speedup obtained by compiling the set of benchmarks with the selected set of optimizations, relative to the execution time resulting from using `-O0`. Again, this does not affect the generality of the COLE framework; COLE can be (trivially) used for other (additional) objective functions of interest, such as code size, energy consumption, power consumption, etc. We stick to just two common objective functions to ease the presentation of the results.

Besides these objectives, we also take the number of enabled optimizations into account; whenever we run into an optimization level L that yields the same generated code but has fewer optimizations enabled compared to another optimization level, we consider the former level to be Pareto dominant. This results in a more focused exploration by the COLE framework: gradually, more and more emphasis is put onto the optimizations that make the biggest difference in terms of the objective functions. By consequence this allows for analysis of the optimizations used in the final Pareto frontier, since the vast majority of optimizations that are used in the levels part of the frontier are the most effective ones. This would not be the case without this additional dominance constraint.

5.1.3 Exploration speed

Exploring the optimization level design space using COLE requires that each entity is evaluated in order to assess its Pareto optimality. Evaluating an entity requires quantifying the entity in terms of the objective functions of interest. In our work, this requires compiling the benchmarks using the set of compiler optimizations that the entity represents. Quantifying code size can be done easily by determining the size of the resulting binary. If performance and/or energy consumption are of interest as objective functions, the compiled benchmarks need to be run, and execution time and energy consumption need to be measured, either through simulation or real hardware execution.

Evaluating an individual entity may be time-consuming. However, evaluating a generation of entities as a whole is embarrassingly parallel. All entities in a generation — there are 20 entities per generation in our setup — can be evaluated in parallel. Subsequent generations need to be run sequentially though because the next generation is computed based on the previous generation.

5.2 Evaluation and analysis

In this section, we present our experimental results which confirm that the automatic construction of compiler optimization levels is feasible in practice. In addition, we show that the levels produced by COLE outperform those obtained through random search, as well as GCC's (manually derived) standard optimization levels for the set of benchmarks used in our experiments. We also analyze the Pareto optimal optimization levels obtained in order to gain insight into the importance of the various compiler optimizations.

5.2.1 Experimental setup

Benchmarks

To evaluate the COLE framework, we use the SPEC CPU2000 benchmark suite (see Appendix B.4). Evaluating one optimization level requires compiling and running all the benchmarks. Compiling and running the benchmarks using the train inputs takes about 10 minutes in our setup. As mentioned before, evaluating a generation of entities can

be trivially parallelized by evaluating all entities in that generation in parallel. Thus, if enough systems are available, evaluating one generation as a whole only takes about 10 minutes in our setup. Nevertheless, speeding up the search process, for example by limiting the number of optimization levels that need to be evaluated, is an interesting avenue for future research.

Compiler and optimizations

We use the GNU Compiler Collection (GCC) 4.2.4 compiler, and consider all the individual compiler optimizations appearing in the standard `-O1`, `-O2`, `-O3` and `-Os` compiler optimization levels that can be turned on and off individually through command-line switches, along with two additional optimizations (`-ftree-vectorize` and `-fschedule-insns`).

There are 68 compiler optimizations in total. The corresponding compiler flags are shown in Table 5.1 which are ordered according to their inclusion in the standard `-O1`, `-O2` and `-O3` compiler optimization levels. For example, the `-O1` optimization level includes all the optimizations listed in the left column of Table 5.1, along with the top 10 ones in the right column; `-O2` includes all optimizations in the left column plus all the optimizations in the right column from the top down to and including `-ftree-vrp`; `-O3` includes all optimizations in Table 5.1 except for the last two; `-Os` includes all optimizations used in `-O2`, except those indicated by '–' in Table 5.1, and adds one additional flag marked with '+', `-finline-functions`.

Next to these 68 compiler optimization flags, we also consider a base optimization level with four possible options, `-O1-stripped`, `-O2-stripped`, `-O3-stripped` and `-Os-stripped`. These stripped base optimization levels correspond to their respective standard compiler optimization levels `-O1`, `-O2`, `-O3` and `-Os` with all optimizations disabled that are controllable through command-line switches; disabling a standard optimization `-f<flag>` can be done using the `-fno-<flag>` command-line switch. The reason for including these stripped base optimization levels is that GCC includes some default optimizations that cannot be controlled through command-line compiler switches, and that several optimizations are only activated if the base level is set high enough, e.g., "`if(level > 1)`".

Table 5.1: The list of compiler optimization flags considered in this chapter, grouped by standard optimization level they are used in. Optimizations used in `-O1` are also used in `-O2` and `-O3`; likewise, optimizations used in `-O2` are also used in `-O3`. All optimizations used in `-O2` are used in `-Os`, except for the ones marked with '–'. Optimizations marked with '+' are used in `-Os`, but not in `-O2`.

1	-falign-loops		35	-ftree-ch (–)	
2	-fbranch-count-reg		36	-ftree-copy-prop	
3	-fearly-inlining		37	-ftree-copyrename	
4	-ffunction-cse		38	-ftree-dce	
5	-fgcse-lm		39	-ftree-dominator-opts	`-O1`
6	-finline		40	-ftree-dse	(cont.)
7	-finline-functions-called-once		41	-ftree-fre	
8	-fivopts		42	-ftree-sink	
9	-fkeep-static-consts		43	-ftree-sra	
10	-fmove-loop-invariants		44	-ftree-ter	
11	-fpeephole		45	-fcaller-saves	
12	-freg-struct-return		46	-fcrossjumping	
13	-fsched-interblock		47	-fcse-follow-jumps	
14	-fsched-spec	`default`	48	-fdelete-null-pointer-checks	
15	-fsched-stalled-insns-dep		49	-fexpensive-optimizations	
16	-fsplit-ivs-in-unroller		50	-fgcse	
17	-ftop-level-reorder		51	-foptimize-register-move	
18	-ftree-loop-im		52	-foptimize-sibling-calls	
19	-ftree-loop-ivcanon		53	-fpeephole2	`-O2`
20	-ftree-loop-optimize		54	-fregmove	
21	-ftree-vect-loop-version		55	-freorder-blocks (–)	
22	-fzero-initialized-in-bss		56	-freorder-functions	
23	-maccumulate-outgoing-args		57	-frerun-cse-after-loop	
24	-malign-stringops		58	-fschedule-insns2	
25	-mno-push-args		59	-fstrict-aliasing	
26	-fno-unit-at-a-time		60	-fstrict-overflow	
27	-fcprop-registers		61	-fthread-jumps	
28	-fdefer-pop		62	-ftree-pre (–)	
29	-fguess-branch-probability		63	-ftree-vrp	
30	-fif-conversion	`-O1`	64	-fgcse-after-reload	
31	-fif-conversion2		65	-finline-functions (+)	`-O3`
32	-fmerge-constants		66	-funswitch-loops	
33	-fomit-frame-pointer		67	-ftree-vectorize	
34	-ftree-ccp		68	-fschedule-insns	

Hardware platforms

We evaluate the COLE framework using two different hardware platforms: systems with an Intel Xeon L5420 processor, which implements the Intel Core microarchitecture, and systems with an Intel L5520 processor, which implements the Intel Nehalem microarchitecture. For more details, we refer to Appendix C.1. For convenience, we will refer to these systems as the Core and Nehalem systems, respectively.

COLE setup

In the initial generation of the COLE optimization process, we include levels with all flags disabled, and levels with all flags enabled, next to a number of randomly generated optimization levels, in order to give the multi-objective search algorithm a head start in its exploration.

We consider one single population, 20 entities per population, and 20 entities per archive in our experiments. The algorithm selects the Pareto optimal entities from the current population and the prior archive, and fills the archive with either a selection of Pareto optimal entities covering the entire Pareto frontier, or all Pareto optimal entities supplemented with the best non-Pareto optimal entities. The mutation probability is set to 0.15; the probability for crossover is set to 0.85. We use a crossover mixing rate of 0.25, and the value of the multi-point drift mutation control parameter is set to 0.25. Convergence is detected when no change is observed in the Pareto frontier for three subsequent generations. These settings were obtained empirically following common practice in evolutionary algorithms, and we found these settings to work well in our setup.

5.2.2 Evaluation

Pareto frontiers

First, we evaluate the quality of the obtained Pareto optimal optimization levels by visual inspection of the Pareto frontiers. Figures 5.2 and 5.3 show the Pareto frontiers obtained on the Core and the Nehalem systems, respectively. Each graph also shows the reference point used for quantifying the quality of each Pareto frontier. The reference point corresponds to the point obtained by taking the minimum or maximum value observed along each objective function (depending on whether the metric is higher-is-better or lower-is-better).

Each figure shows Pareto frontiers in terms of compilation cost and speedup obtained in three different ways:

- the standard GCC optimization levels (`-O1`, `-O2`, `-O3` and `-Os`); the optimization level `-Os` is not a Pareto optimal optimization level (in terms of the compilation cost and speedup objective functions);

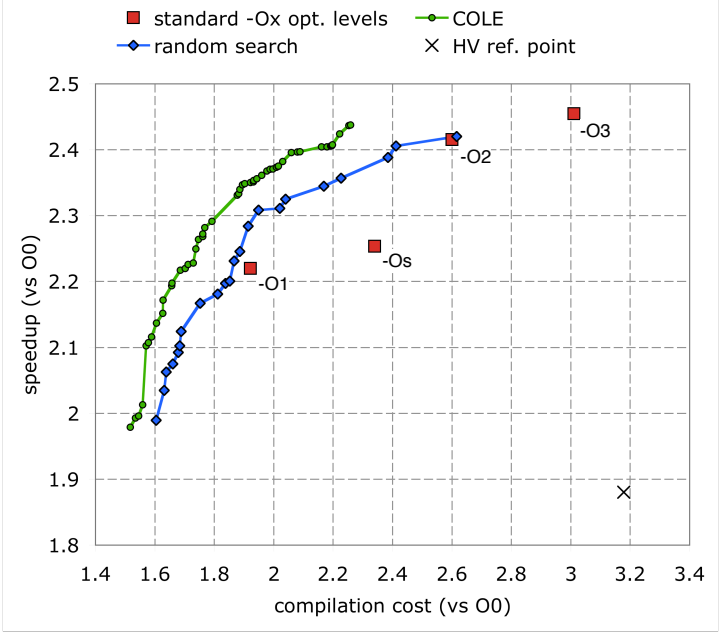


Figure 5.2: Pareto frontier containing candidate optimization levels trading off speedup and compilation cost, obtained using COLE on the Core systems and the SPEC CPU2000 benchmarks.

- the Pareto frontier obtained by randomly sampling the optimization space: 10,000 randomly chosen optimization levels were evaluated and the Pareto optimal optimization levels are retained;
- the Pareto frontier obtained through COLE after convergence. On the Core systems, 208 generations were required for convergence, during which 3,628 unique entities were evaluated; the exploration on the Nehalem systems took 187 generations to converge, during which 3,400 unique entities were evaluated.

Thus, COLE required about 36,280 minutes (3,628×10 min.) or 605 hours on the Core systems, and 34,000 minutes (3,400×10 min.) or 567 hours on the Nehalem systems. Taking into account that each generation can be evaluated in parallel, this can be limited to just 2,080 minutes (35 hours) and 1,870 minutes (31 hours), respectively, if enough (identical) systems are available.

The results in Figures 5.2 and 5.3 show that the Pareto frontier

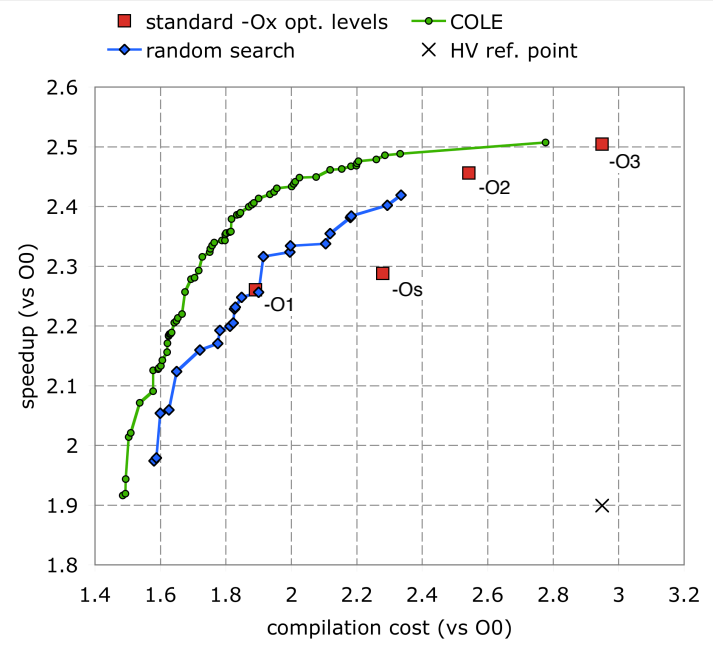


Figure 5.3: Pareto frontier containing candidate optimization levels trading off speedup and compilation cost, obtained using COLE on the Nehalem systems and the SPEC CPU2000 benchmarks.

obtained through COLE is substantially better in terms of compilation cost than the standard GCC optimization levels. For example, roughly the same average speedup can be obtained compared to `-O2` at a substantially lower compilation cost, i.e., 16.9% and 18.3% lower on the Core systems and Nehalem, respectively. Compared to `-O1` roughly the same code quality (quantified in terms of speedup) can be achieved at a lower compilation cost, and a higher speedup is obtained at roughly the same compilation cost (6.0% higher on the Core systems and 6.5% higher on the Nehalem systems). On the Core systems, the right-most COLE optimization level obtains an average speedup 0.7% lower compared to `-O3`, but does so with a compilation cost that is 15.0% lower; on the Nehalem systems, the best optimization level in terms of speedup obtained with COLE yields a speedup 1.0% higher than `-O3`, at a compilation cost which is 5.8% lower.

Comparing the Pareto frontier obtained with COLE with the frontier obtained through random sampling reveals that COLE explores the search space in a more effective way and obtains better results that way,

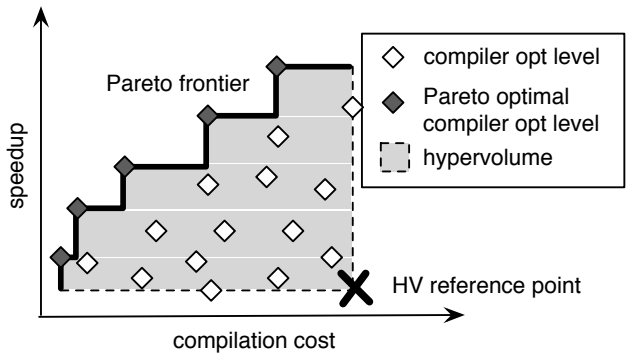


Figure 5.4: Example Pareto frontier, illustrating how the hypervolume (HV) metric is computed.

even with a significantly lower search budget. Both on the Core and Nehalem systems, COLE yields a Pareto curve which clearly outperforms the one obtained by evaluating roughly 2.5 times as many randomly picked optimization levels.

It should be noted that these results are tied to the particular set of benchmarks used, and the platforms on which the benchmarks are compiled and executed. It is no surprise that the best optimization levels for a particular set of benchmarks on a particular platforms obtained through random search and COLE significantly outperform the standard optimization levels readily available in GCC; these levels are tuned for a much larger set of applications and a broader range of hardware platforms. Nevertheless, the fully automated COLE framework is very useful to both compiler developers and end users, because it is able to identify a diverse set of optimization levels that could serve as standard optimization levels, and does so in a more effective way than simple random searching.

Quantitative comparison

We now also evaluate the quality of the Pareto frontiers obtained through COLE using the hypervolume (HV) metric discussed in [30].

Figure 5.4 illustrates how the HV metric is computed. After determining the HV reference point, the hypervolume can be computed as the area between the Pareto frontier and the HV reference point. The HV metric thus is a higher-is-better metric. Note that although we only

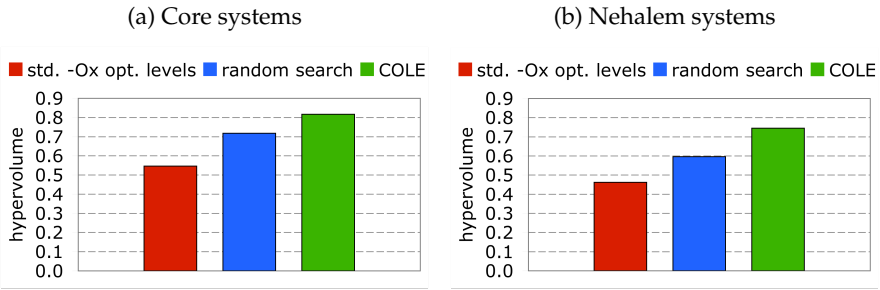


Figure 5.5: Quantification using the hypervolume metric of the standard optimization levels and the Pareto optimal optimization levels obtained through random search and using the COLE framework, on the Core and Nehalem systems.

use the HV metric with two objective functions, it is straightforward to use the same metric with more than two objective functions of interest.

Figure 5.5 compares the values of the HV metric for the standard optimization levels available in GCC and the Pareto frontiers obtained using random search and COLE. These graphs quantitatively confirm the conclusions drawn from the visual inspection of the Pareto frontiers in the previous section.

Both for the Core and Nehalem systems, the standard optimization levels are significantly outperformed by both Pareto frontiers; the HV metric for the COLE Pareto frontier is 49.4% and 61.2% higher, on both hardware platforms, respectively. The COLE Pareto frontier is 13.8% and 25.0% better, respectively, than the frontier obtained through random search in terms of the HV metric, which is a substantial difference given the large difference in search budget.

Cross-validation

So far, we evaluated our COLE framework with the SPEC CPU2000 benchmarks, which is the same set of benchmarks used to construct the Pareto optimal optimization levels. Likewise, we evaluated the optimization levels on the same hardware platform as the one on which the COLE framework performed its measurements to evaluate optimization levels. In this section, we consider two cross-validation experiments in which we evaluate the cross-application and cross-platform effectiveness of the Pareto optimal optimization levels obtained using

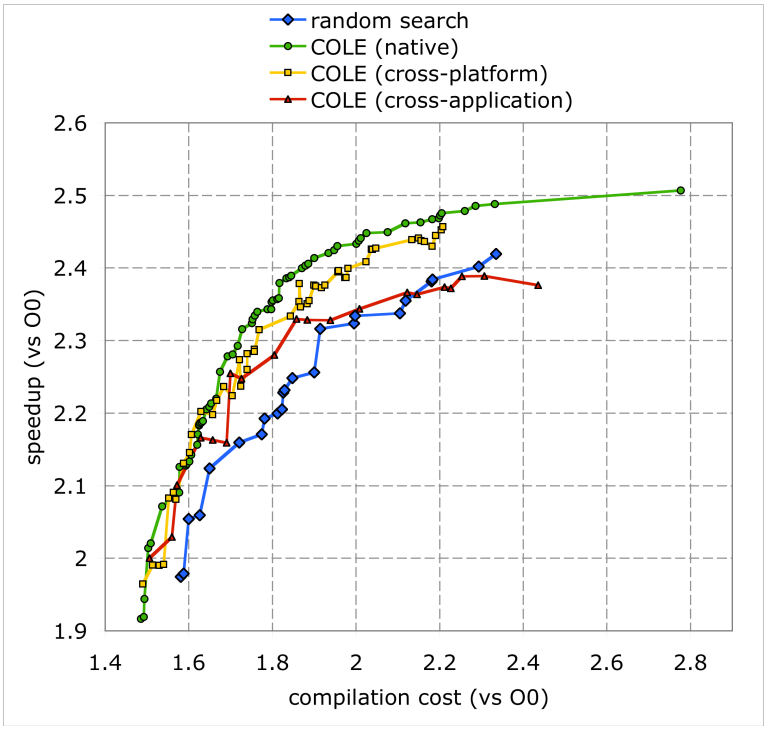


Figure 5.6: Experimental results of two cross-validation experiments on the Nehalem systems: the cross-platform Pareto curve is obtained by evaluating optimization levels constructed for the Core systems, while the cross-application Pareto curve is constructed using optimization levels targeted to the SPEC CPU2006 benchmark suite (as opposed to the SPEC CPU2000 benchmarks).

our COLE framework. The results of these cross-validation experiments are shown in Figure 5.6.

Cross-application evaluation First, we look at the effectiveness of optimization levels obtained for a different set of applications. Through COLE, we obtained a set of Pareto optimal optimization levels for the SPEC CPU2006 benchmark suite[4]. Evaluating these levels using the SPEC CPU2000 benchmarks allows assessing the cross-application effectiveness of the optimization levels obtained with COLE, see Figure 5.6.

[4] Again, we used the train inputs when running the benchmarks.

The optimization levels obtained for the SPEC CPU2006 benchmarks are clearly suboptimal in terms of compilation cost and speedup obtained for the SPEC CPU2000 benchmarks; the right-most CPU2006 optimization levels are even outperformed by the ones obtained for CPU2000 by means of random search. More quantitatively, the Pareto frontier for the CPU2006 optimization levels is 18.7% lower than CPU2000 Pareto frontier obtained with COLE in terms of the HV metric; compared to the random Pareto frontier, it is 14.5% better. While the CPU2006 optimization levels that focus on low compilation cost are competitive with the CPU2000 levels, the ones targeted towards high speedup are inferior to the CPU2000 levels. In particular, the CPU2006 optimization level that performs best in terms of speedup for the CPU2000 benchmarks, i.e., the second level from the right, yields code that is 3.6% slower than the CPU2000 optimization level which is most competitive in terms of compilation cost; likewise, the compilation cost of the CPU2000 optimization level that yields a similar speedup as this CPU2006 level is 22.4% lower.

Cross-platform evaluation A second cross-validation experiment evaluates the effectiveness of optimization levels obtained for one particular hardware platform when they are being used on a different hardware platform. For this experiment, we evaluate the Pareto optimal optimization levels obtained through COLE for the Core systems on the Nehalem systems. For convenience, we will refer to the optimization levels constructed for the Nehalem systems as the *native* optimization levels, and the ones obtained on the Core systems as the *cross-validation* optimization levels (see Figure 5.6).

The cross-validation optimization levels are fairly competitive with the native optimization levels; in terms of hypervolume, the cross-validation Pareto frontier is just 6.4% lower than the native Pareto frontier. In some cases, one of the cross-validation levels even outperforms some of the native optimization levels. Considering the entire range of trade-offs however, the native optimization levels clearly have the upper hand, especially for levels that deliver the highest speedups. The right-most cross-validation optimization level yields a speedup 0.8% lower than the native optimization level with roughly the same compilation cost, and 2.2% lower than the right-most native optimization level.

Although the native and cross-validation Pareto curve are fairly

competitive in this particular case, this will probably not be the case in general. Relatively speaking, microprocessors based on the Core microarchitecture are fairly similar to microprocessors that implement the Nehalem microarchitecture: same ISA (Intel 64-bit), comparative number of registers, similar cache memory hierarchies, etc. Performing a similar experiment using systems which differ in more ways may show more significant differences, in favor of the native optimization levels.

Conclusions The results of the cross-application and cross-platform experiments support our initial motivation for an automated framework for constructing compiler optimization levels. The experiments discussed above show that tuning for a particular hardware platform and a particular set of applications can have significant benefits in terms of the trade-off between compilation cost and application performance. Currently, compiler developers are forced to resort to manually developing a generic set of optimization levels. Besides the fact that this a labor-intensive and thus time-consuming task, our experiments show that it is likely that a significant price is paid for this generality. An automated framework like COLE is an important step in supporting the construction of more specialized optimization levels, both for a particular hardware platform and a particular application domain.

5.2.3 Analysis

In order to analyze the Pareto optimal optimization levels obtained through COLE in terms of their constituent compiler optimizations, we show the composition of the optimization levels in Figure 5.7 (Core systems) and Figure 5.8 (Nehalem systems). In these figures, each row of boxes corresponds to one optimization level, while each column, except for the first one, corresponds to one particular compiler optimization. The first column indicates the base optimization level used (`-O1`, `-O2`, `-O3` or `-Os`). The compiler optimizations are shown in the same order (left to right) as they are listed in Table 5.1, and can be identified using the index number shown in that same table. The optimization levels are ordered horizontally from low compilation cost and low speedup (left-most levels in Figures 5.2 and 5.3) to high compilation cost and high speedup (right-most levels). A filled box indicates that the corresponding compiler optimization is enabled for that particular optimization level.

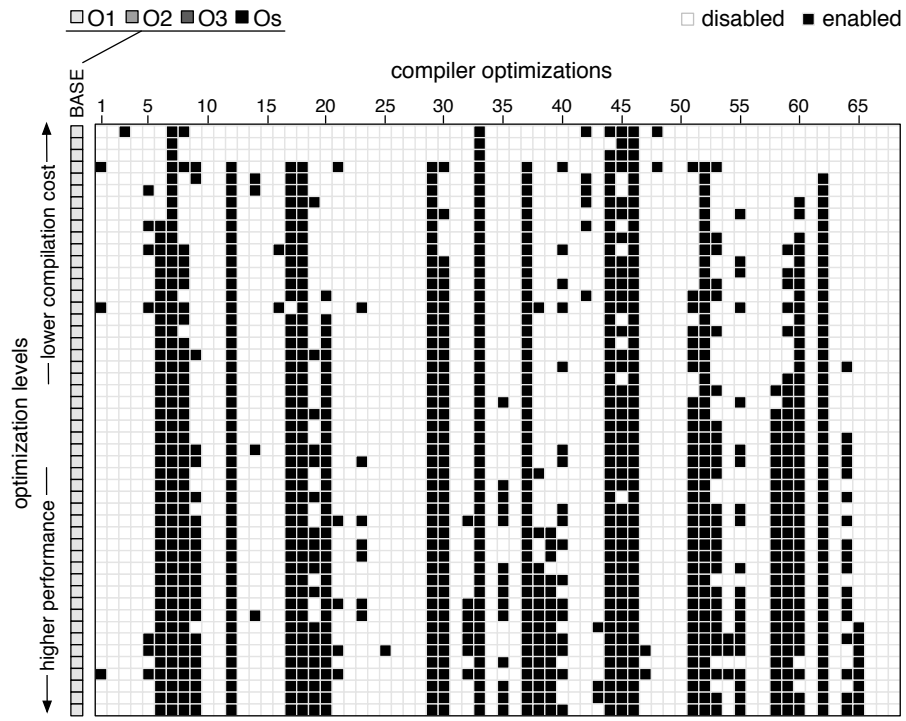


Figure 5.7: Composition of the 50 Pareto optimal optimization levels obtained for the Core systems and SPEC CPU2000 (train). Each row represents one optimization level (horizontally sorted from lowest compilation cost and speedup to highest compilation cost and speedup), each column represent one of 68 compiler optimizations used in this study (see Table 5.1).

A number of interesting observations can be made from these figures. First, some compiler optimizations are never used, regardless of the trade-off made between compilation cost and speedup. In the optimization levels obtained for the Core systems, 24 compiler optimizations are never used; for the Nehalem systems, 9 levels are never used (19 if we omit the last optimization level). In other words, 35% of the compiler optimizations are not used on the Intel Core systems; likewise, 28% of the optimizations are never used in any of the optimization levels (except the last one) on the Nehalem systems. Four compiler optimizations are never used at all, on either of both hardware platforms: `-fno-unit-at-a-time`, `-fcprop-registers`, `-fexpensive-optimizations` and `-fschedule-insns`. There are various reasons why a particular compiler optimization might not

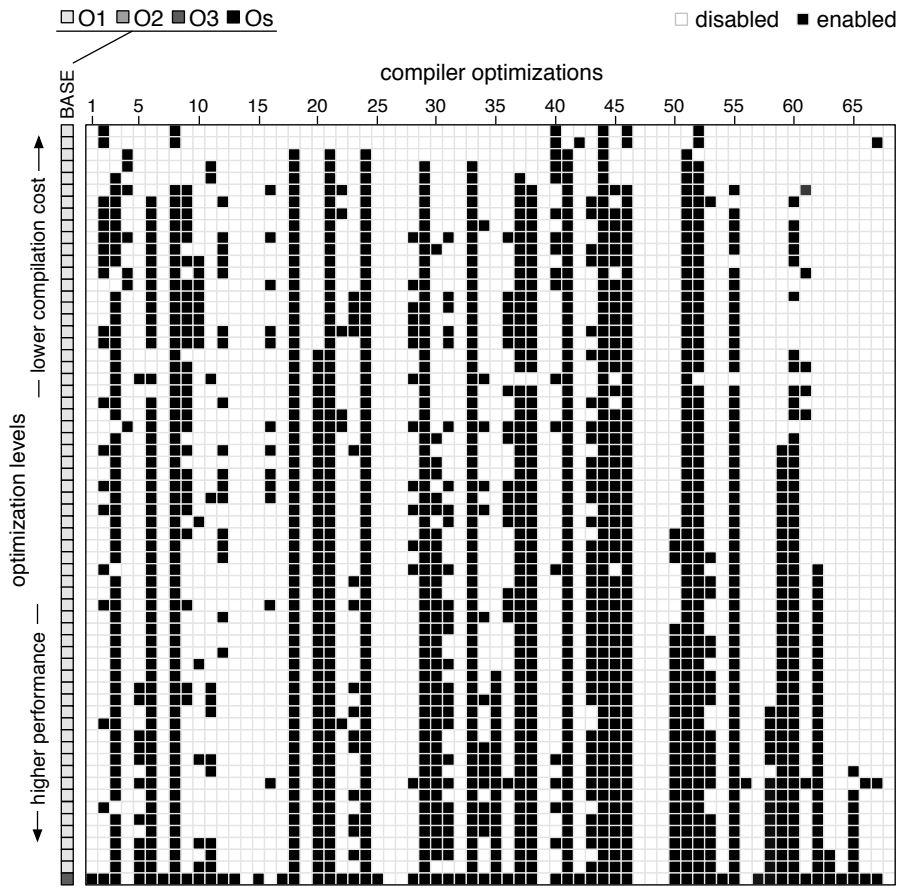


Figure 5.8: Composition of the 64 Pareto optimal optimization levels obtained for the Nehalem systems and SPEC CPU2000 (train). Each row represents one optimization level (horizontally sorted from lowest compilation cost and speedup to highest compilation cost and speedup), each column represent one of 68 compiler optimizations used in this study (see Table 5.1).

be enabled. A non-extensive list of possible reasons includes:

- the optimization is not applicable to the particular set of target applications; enabling the optimization would result in additional analysis being performed, adding up to the overall compilation cost, but without any benefit in terms of performance;
- the optimization is too expensive in terms of compilation cost (e.g., too intrusive analysis is required), and does not yield a significant gain in speedup to justify the added cost;

- the optimization is interfering with other optimizations, for example by removing opportunities for other optimizations that are beneficial either in terms of compilation cost or speedup.

Of course, this is tied to the particular hardware platforms used in our experiments, and more importantly also to the SPEC CPU2000 benchmark suite which the COLE framework used to evaluate each optimization level.

Some compiler optimizations come forward as critical, i.e., they are used in a large number of Pareto optimal optimization levels: 11 and 13 optimizations are enabled in at least 90% of the optimization levels, on the Core and Nehalem systems, respectively. Overall, 7 compiler optimizations are used in at least 90% of the Pareto optimal optimization levels on both platforms: `-free-loop-im`, `-fguess-branch-probability`, `-fomit-frame-pointer`, `-ftree-copyrename`, `-ftree-ter`, `-fcrossjumping` and `-foptimize-sibling-calls`. Apparently, the corresponding optimizations are beneficial regardless of the trade-off between compilation cost and speedup.

When we compare the composition of the Pareto optimal optimization levels on both hardware platforms, there are some remarkable differences. For example, the `-ftree-fre` optimization is enabled in 62 of the 64 optimization levels on the Nehalem systems, but is never used in any of the optimization levels obtained for the Core systems. It seems that the corresponding Full Redundancy Elimination analysis which attempts to remove redundant computations is too expensive in terms of compilation cost on the Core systems to be considered beneficial, while this is not the case on the Nehalem systems. Similar observations include `-finline-functions-called-once`, which is always used on the Core systems and is only enabled for the last optimization level (highest compilation cost and speedup) on the Nehalem systems, and `-malign-stringops`, which appears to be critical for the Nehalem systems, but is never used on the Core systems. These observations confirm that the construction of optimization levels should be done specifically for the hardware platform on which they will be used in order to obtain the best results; an automated framework as proposed in this chapter is thus very desirable in this regard.

We showed that analysis of the Pareto optimal optimization levels obtained using our COLE framework is likely to yield interesting observations for compiler developers. These observations can steer further

efforts to improve existing compiler optimizations, and to study the interactions that occur between different optimizations. Some of the observations made above would have been more difficult to obtain without a fully automated framework for constructing optimization levels, in particular because a compiler developer manually constructing optimization levels heavily depends on his intuition and experience.

5.2.4 Discussion

Although the results presented in this section show that COLE is indeed a useful tool for automatically constructing optimization levels, there is one major drawback. Even though the number of candidate optimization levels that need to be evaluated is very limited compared to the enormous size of the search space, the total amount of time required to evaluate the levels considered limits the applicability of the framework. While sufficient computing resources can help resolve this problem by evaluating each generation of optimization levels in parallel, a significant reduction of the exploration time can be achieved by either lowering the time required to evaluate one particular optimization level (for example by relying on partial evaluation), or by limiting the number of levels that actually need to be evaluated. One promising avenue for future work is the field of *global optimization* [65]. Applying this technique would require to gradually build more and more accurate predictive models for each of the objective functions, which can then be used to guide the evolutionary search algorithm without requiring evaluation of each of the optimization levels being considered.

5.3 Related work

The work most closely related to the work presented in this chapter is iterative compilation. The basic idea of iterative compilation is to explore the compiler optimization space by iteratively compiling and measuring the effectiveness of a set of optimizations. Driven by a search algorithm, iterative compilation explores the optimization space, and upon termination of the search algorithm, the best performing set of optimizations is reported for the given application. A large body of work has been done on iterative compilation over the past few years, and many researchers have reported impressive results showing significant performance, energy or code

size improvements over standard optimization levels, see for example [1, 3, 17, 19, 26, 28, 42, 45, 49, 68, 83, 98, 104, 110].

An important concern though with iterative compilation is that searching the optimization space is very time consuming. By consequence, the vast majority of the work on iterative compilation focuses on reducing the search time; there are basically two ways for doing so. One approach is to speed up the search process by either pruning the search space [98], or intelligently navigating through the search space using heuristic search algorithms, such as genetic algorithms [28, 68] or combination elimination [83]. Another approach is to reduce the time spent evaluating a design point during this search. Some researchers propose analytical modeling for estimating the effect of compiler optimizations on performance [98, 110]. Others build empirical models using predictive modeling built from static code features [1, 21, 104] or dynamic code features [19]. Yet others exploit the phase behavior observed during application execution, and evaluate different optimization sequences in subsequent occurrences of the same phase [42].

What all of this prior work on iterative compilation has in common is that it focuses on a single objective function to be optimized. For example, researchers typically focus on a single optimization criterion such as performance [1, 17, 19, 21, 28, 42, 47, 83, 98, 104, 110], or energy consumption [45], or code size [28]. And some researchers focus on optimizing a single objective function that combines multiple optimization criteria such as code quality and compilation time [20, 21], or code quality and code size [68]. All of these approaches optimize a single metric. And this is where the key difference lies between the prior work on iterative compilation and the COLE approach described in this paper. COLE aims at exploring a multi-objective compiler optimization space, whereas prior work is limited to single-objective optimization. Or, in other words, COLE yields multiple Pareto optimal points whereas prior work yields a single optimal point. An additional difference between existing iterative compilation work and COLE is that iterative compilation focuses on optimizing the performance of a single application, whereas COLE allows finding compiler optimization levels that improve the average performance for a collection of applications.

5.4 Summary

Compilers typically come with a number of optimization levels such as `-O1`, `-O2`, `-O3` and `-Os`, which provide different trade-offs between code quality, compilation time and code size. Constructing these optimization levels typically is a manual process which is both tedious and time consuming. Identifying an appropriate set of optimization levels is particularly challenging because the search space is huge — for example, the design space in our setup counts in the order of 10^{20} candidate optimization levels.

In this chapter, we presented COLE, Compiler Optimization Level Exploration, which employs a multi-objective evolutionary algorithm to find Pareto optimal optimization levels. COLE is fully automated and is completely transparent to the compiler, the benchmarks, the hardware platform, as well as the objective functions. To the best of our knowledge, this work is the first to study automated multi-objective compiler optimization exploration. COLE differs from iterative compilation in this respect because the work done so far in iterative compilation focused on single-objective optimization.

Our experiments using GCC and the SPEC CPU benchmarks on both an Intel Core and Intel Nehalem system optimizing for run time (code quality) and compilation cost show that the optimization levels obtained through COLE significantly outperform the standard (and manually derived) compiler optimization levels (`-O1`, `-O2`, `-O3`), and in addition, outperform the optimization levels obtained through random sampling. The cross-validation experiments show that targeting the optimization levels to the particular hardware platform and application domain for which they will be employed is important with respect to obtaining good trade-offs. A thorough analysis of the composition of the optimization levels obtained through COLE leads to interesting observations about the contribution of the different optimizations; a significant amount of optimizations are not used, regardless of the trade-off between code quality and compilation cost, while other optimizations come forward as critical because they are beneficial overall.

Chapter 6

Automated Just-In-Time Compiler Tuning

Make everything as simple as possible, but not simpler.
Albert Einstein

One of the key advantages of managed programming languages, such as Java, is that programs are compiled to an intermediate machine-independent level, called bytecode, enabling cross-platform portability. However, this requires a process virtual machine—a Java virtual machine or JVM for short—to translate bytecode to executable code. Modern JVMs tend to follow a mixed-mode execution scheme in which application methods are first interpreted, or compiled with a baseline non-optimizing compiler. If a method is sufficiently *hot*, i.e., is executed frequently, it will likely be a candidate for (re)compilation by the optimizing Just-In-Time (JIT) compiler. In this chapter, we refer to a set of optimizations used together during the (re)compilation of a method as an *optimization plan*. Modern JVMs [5, 76, 82] employ multiple *optimization levels* (e.g., `-O0`, `-O1` and `-O2`), in which each level comprises a successively more aggressive optimization plan. In other words, more aggressive optimizations are performed on more frequently executed code: higher optimization levels result in longer compilation times, yet they supposedly yield better code, thereby further speeding up the execution of the hot methods.

Tuning the VM's JIT compiler is a challenging task for a number of reasons. For one, to ensure good performance, the VM developer has to carefully pick and tune each of the optimization levels, choosing the right optimizations at each level and tweaking their settings and con-

trols. As mentioned during the previous chapter, this is far from trivial because of the large number of available optimizations and their complex interactions. Second, the Adaptive Optimization System (AOS), i.e., the engine that decides which methods to optimize to which optimization level, needs to be fine-tuned. This is non-trivial as well because the optimum AOS configuration is highly dependent on the optimization plans at each optimization level and it is crucial to take full advantage of the available optimization levels. Third, this tuning process needs to be done for every possible optimization target of interest. In particular, the optimal VM configuration may be specific to a particular hardware platform because different hardware platforms come with different memory hierarchies, microarchitectures, etc. which requires the JIT compiler to be tuned differently. Different applications may need the JIT compiler to be tuned differently as well. For example, servers often run a single application or a limited number of applications, such as middle-ware or business applications, over and over again. As such, it makes sense to tune the VM for a particular application or set of applications.

Current practice is to manually tune the JIT compiler. Arnold et al. [7] and Ishizaki et al. [62] describe such a manual process for the Jikes RVM and the IBM JDK production VM, respectively. This process is both tedious, time-consuming and costly, and may lead to suboptimal performance. Moreover, tuning needs to be done for every new processor on the market as well as for different applications and application domains.

In this chapter, we propose automated JIT compiler tuning. This is done in two steps. The first step identifies optimization plans that are Pareto optimal in terms of compilation time and code quality. This is done using the COLE framework presented in Chapter 5. We subsequently retain a limited number of optimization plans that cover the Pareto frontier well. The second major step is to search for the optimum JIT compiler. This involves assigning Pareto optimal optimization plans to optimization levels (`-O0`, `-O1` and `-O2`), and fine-tuning the AOS. Again, we use machine learning techniques, and evolutionary search algorithms in particular, for doing so. The end result is a VM that is optimized for the optimization target(s) of interest, i.e., for a given hardware platform and/or application domain.

To the best of our knowledge, we are the first to propose a framework for automatically tuning a JIT compiler with multiple optimiza-

tion levels for optimum performance. The key benefit is that the exploration is fully automated and enables tuning the JIT compiler for a given hardware platform and/or (set of) application(s) at very low cost.

6.1 Java Virtual Machine: Jikes RVM

Before presenting the proposed JIT compiler optimization framework, we first briefly describe the organization of a modern Java virtual machine, namely Jikes RVM (Research Virtual Machine) [5]. This will enable us to better understand the complexity of JIT compiler tuning.

Although this work uses the Jikes RVM for driving the experiments, we strongly believe that the overall framework and conclusion is applicable to other Java virtual machines. Moreover, similar JIT compiler tuning can be applied on other process virtual machines, such as the Common Language Runtime (CLR) of the Microsoft's .NET initiative.

6.1.1 Optimization plans and levels

JikesRVM is a compilation-only VM. Methods are initially compiled using a fast but non-optimizing baseline compiler that generates relatively inefficient machine code. To improve performance, Jikes RVM employs a JIT optimization strategy for optimizing hot methods using three optimization levels (`-O0`, `-O1` , and `-O2`). We refer to the baseline compilation level as `base`.

Each optimization level `-On` is defined by an optimization plan $P_{\texttt{On}}$ that enumerates the optimizations at that level along with several values that further steer their use. In the default Jikes RVM configuration, optimization plans for higher levels include the optimizations for the lower levels. Each optimization level also has a corresponding *aggressiveness* assigned to it that influences the use of various optimizations, e.g., more copy propagation passes are done at higher optimization levels. In Jikes RVM (version 3.0.1), there are 33 boolean options available, each of which turns an optimization on or off, and 10 value options that control the optimizations[1]. Thus, per optimization plan, we have 2^{33} possible combinations of boolean flags and a space spanned by eight

[1]These are the options we have used in the exploration. There are other options we did not use because they are either unstable, not meant to be changed from outside the VM or can activate options that result in breaking the Java language specification.

Table 6.1: Default compiler DNA values for Jikes RVM v3.0.1.

	`base`	`-O0`	`-O1`	`-O2`
compilation rate (bc/ms)	909.46	39.53	18.48	17.28
speedup vs. `base`	1.0	4.03	5.88	5.93

positive integer values and two positive floating-point values. This results in a huge search space.

6.1.2 Compiler DNA

An optimization plan is characterized using two metrics: the compilation rate (i.e., bytecodes compiled per millisecond (bc/ms)), and the improvement in code quality (i.e., speedup in execution time over `base`). Combined, these two metrics are referred to as the *compiler DNA* associated with the optimization plan.

The compiler DNA for each optimization plan/level in Jikes RVM is measured as follows. The compilation rate is obtained by compiling *all* methods at the specified optimization level upon first execution. The speedup is the ratio between the execution time obtained by executing this optimized code and the execution time for a VM using the `base` compiler only. The DNA in Jikes RVM for x86, see Table 6.1, was computed on an LS41 type 7972 blade, equipped with an AMD Opteron 8218 with 4 MB L2 cache and 4 GB RAM, using the SPECjvm98 benchmarks[2].

6.1.3 Sample-based JIT optimization

Jikes RVM uses OS-timer triggered sampling to identify hot methods. When the timer fires, the method on top of the stack is sampled the moment a yield point[3] is reached [7, 8]. When sufficient samples have been gathered for a method, the VM uses the AOS to decide whether or not to optimize the method to a particular optimization level.

[2]The Jikes RVM compiler DNA for the PowerPC platform specifies different values.

[3]A yield point in Jikes RVM is a point during the execution where the scheduler can safely switch threads. It is placed at the beginning and the end of methods and at loop back-edges.

6.1.4 Adaptive Optimization System

The AOS decides whether or not to optimize a method, and if so, to which optimization level the method should be optimized. There are five value options in total that control the AOS: three positive integer values, and two positive floating-point values, again resulting in a large space to explore. The AOS parameters control when the engine finds a method to be hot enough to be considered for optimization to a higher level. It uses the compiler DNA to make a trade-off in compilation cost (i.e., how long does it takes to optimize the method at a given optimization level?) and code quality (i.e., how much faster will the code run once optimized?).

6.2 Methodology

We now present our framework for automatically tuning a JIT compiler. This includes identifying the optimization plans, optimization levels and AOS settings. Before describing the overall framework in great detail, we first motivate the need for a two-step process.

6.2.1 Why a two-step process?

As mentioned earlier in the introduction, optimizing a dynamic compiler is substantially more complicated than optimizing a static compiler because of the tight interaction between optimization plans and levels, and the AOS settings. For example, including a compiler optimization at one level changes the compilation rate versus code quality trade-off, which in turn changes which methods are optimized to which optimization level. This leads to complex interactions that severely complicate the search process. Our initial approach to this problem was to use an evolutionary algorithm to construct the optimization plans, plan-to-level assignments, the number of optimization levels, and the AOS settings in a single go. In fact, we used the COLE framework [55] presented in Chapter 5 which was developed for a static compiler, and naively applied it to a dynamic compiler by defining an entity that represents a JIT configuration with multiple optimization levels and AOS parameters. However, we encountered three significant problems. First, the automatically derived JIT compiler did not perform as well as or better than the default manually tuned Jikes RVM (and for many

benchmarks performed significantly worse). Second, the search process took extremely long to converge. Third, expressing the optimization problem in a format that can be handled by COLE's evolutionary search algorithm was non-trivial, e.g., it is unclear how to sensibly define crossover across two JIT compiler settings with a different number of optimization levels. This motivated us to come up with a two-step process in which we first focus on code quality versus compilation rate while excluding dynamic compilation and garbage collector (GC) activity, and subsequently assign plans to levels and optimize the AOS settings while considering dynamic compilation and GC activity. The two-step process enables a higher performance JIT compiler to be derived in a shorter amount of time.

6.2.2 Step 1: Pareto optimal optimization plans

Exploration using COLE

The goal of the first step is to identify optimization plans that are Pareto optimal in terms of compilation cost and code quality they deliver. Figure 6.1 shows an illustrative example of a Pareto frontier in the dual-objective search space, namely compilation rate (i.e., number of bytecodes compiled per unit of time) versus speedup (i.e., performance improvement compared to non-optimized code). An optimization plan is Pareto optimal if there is no other plan that performs better both in terms of compilation rate and speedup. When constructing the Pareto frontier, we consider a setup in which we first compile all the code according to the optimization plan and subsequently execute the optimized code—we do not consider JIT compilation (for now) and consider a large heap size (8 times the minimum heap size) to minimize GC activity. This is to understand the basic trade-off in code quality versus optimization overhead.

For identifying the Pareto frontier, we use the COLE framework presented in the previous chapter. We retain all the Pareto optimal optimization plans ever seen during the exploration performed by COLE.

Selecting optimization plans

Through COLE, we obtain a fairly large set of Pareto optimal optimization plans; in our experiments, we obtained up to 80 Pareto optimal plans. From this set, we select a subset such that the Pareto frontier

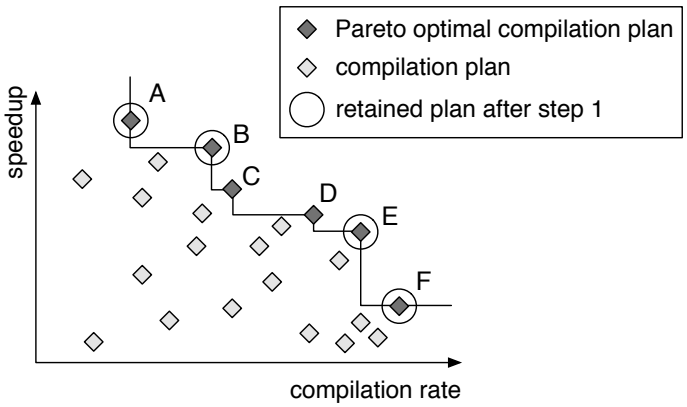


Figure 6.1: An example of a Pareto frontier in our dual-objective exploration space. The circled plans are those retained at the end of the first step to bootstrap the second step.

is covered well. We found this Pareto frontier reduction procedure to be an important step in the overall JIT exploration in order to limit the total exploration time.

The rationale behind the Pareto frontier reduction procedure is to prefer optimization plans that result in higher code quality at roughly the same compilation rate, and compile bytecode faster while attaining roughly the same speedup. We therefore use an iterative selection algorithm. In the first iteration, we pick the two adjacent plans on the Pareto frontier that lie closest to each other along the X axis. We drop the plan that scores worst along the Y axis. We then select the two plans that lie closest to each other along the Y axis, and drop the one that scores worst along the X axis. This iterative process stops when the number of retained plans drops below a given number. We limit the number of retained Pareto optimal optimization plans to 8.

In our running example, see Figure 6.1, this means we first select the pair (B,C) because they lie closest on the X axis and drop C. The next pair is (D, E) because they lie closest on the Y-axis and we only retain E. After two iterations, the set of retained optimization plans equals {A, B, E, F}.

6.2.3 Step 2: JIT compiler tuning

The second step in the proposed JIT compiler tuning framework is to (i) assign the Pareto optimal optimization plans to optimization levels (`-O0`, `-O1` and `-O2`), and (ii) tune the JIT AOS accordingly. In contrast to the first step, we now consider adaptive JIT compilation, i.e., the JIT compiler optimizes the most frequently executed methods at run time, and we consider heap sizes that introduce GC activity in order to achieve representative performance numbers. In other words, compilation and optimization time as well as GC time become part of the overall execution time.

Assigning optimization plans to optimization levels is fairly straightforward: given the limited number of retained Pareto optimal optimization plans we can easily consider all possible assignments of plans to levels. In our setup, this means we need to assign 8 optimization plans to 1 through 3 optimization levels. There are 92 possible assignments. We use a multi-objective evolutionary search algorithm to identify the best AOS settings. In this algorithm, the 92 JIT compilers obtained by assigning plans to levels use the default AOS settings used in Jikes RVM and serve as the initial population. Subsequent populations are then constructed from the best performing JIT compilers (in terms of startup and steady-state performance) through crossover and mutation of the AOS settings, until convergence is reached.

6.3 Evaluation and analysis

In this section, we experimentally evaluate our framework and analyze the results, after detailing on our experimental setup.

6.3.1 Experimental setup

Benchmarks

Table 6.2 shows the benchmarks used in this chapter. We use the SPECjvm98 benchmarks [29] (top seven rows), as well as nine DaCapo benchmarks [15] (bottom nine rows). SPECjvm98 is a client-side Java benchmark suite consisting of seven benchmarks. We run all SPECjvm98 benchmarks with the largest input set (`s100`). The DaCapo benchmark suite is an open-source benchmark suite; we use

Table 6.2: SPECjvm98 (top seven) and DaCapo (bottom nine) benchmarks considered in this paper.

benchmark	description	min heap size (MB)
compress	file compression	24
jess	puzzle solving	16
db	database	32
javac	Java compiler	32
mpegaudio	MPEG decompression	16
mtrt	raytracing	24
jack	parsing	24
antlr	parsing	32
bloat	Java bytecode optimization	56
fop	PDF generation from XSL-FO	56
hsqldb	database	176
jython	Python interpreter	72
luindex	document indexing	32
lusearch	document search	32
pmd	Java class analysis	64
xalan	XML to HTML transformer	40

release version 2006-10-MR2. We include the nine benchmarks that execute properly on the 3.0.1 version of Jikes RVM. The default (`medium`) input set is used for the DaCapo benchmarks unless mentioned otherwise.

Hardware platforms

We use four different hardware platforms in this study:

- an AMD Opteron 242 clocked at 1.6GHz with 1 MB L2 cache and 4 GB RAM running Linux 2.6.9;
- an Intel Pentium 4 clocked at 3.0GHz with 1 M L2 cache and 1.5GB RAM running Linux 2.6.19;
- an Intel Core 2 based Xeon L5420 clocked at 2.5GHz with 6 MB L2 cache and 16 GB RAM running Linux 2.6.18; and
- an Intel Core i7 920 based machine clocked at 2.6GHz with 256 kB L2, 8MB L3 and 12 GB RAM running Linux 2.6.27.

Jikes RVM

We use Jikes RVM version 3.0.1, released on November 18th, 2008. We patched Jikes RVM such that optimizations can be set on a per-optimization level basis at the command line. The virtual machine was built using the *production* profile, which uses the GenMS garbage collector and compiles the VM methods using the optimizing compiler with the default P_{02} optimization plan.

For the optimization plans, we consider the 33 boolean optimization options listed in Table 6.3 in the tuning process, alongside 10 value options. We also consider 5 AOS value options that steer the adaptive compilation. All value options are listed in Table 6.4. For each optimization plan, we consider three base settings (`O0`, `O1` and `O2`), which control the aggressiveness of some optimizations.

During the first step of the exploration algorithm, we use a heap size that is 8 times the minimum size required to run a benchmark; this is to eliminate the effect of garbage collection, as mentioned earlier. We do vary the heap size (i.e., $2\times$, $4\times$, and $8\times$ the minimum heap size) during the second step and during evaluation, following current practice [15].

Tuning framework configuration

As mentioned in Section 6.2.2, we rely on the COLE framework presented in Chapter 5 in the first step of the tuning process. In our setup, each population consists of 25 entities each representing an optimization plan. The archive size is set to 20 entities. Crossover mixing with a crossover rate of 25% is used to construct $9/10$ of the entities in a new generation, while multi-point drift mutation with a mutation rate of 25% is used for the remaining $1/10$ of the entities. Convergence is considered when no change in the Pareto frontier is observed for three subsequent generations.

In the initial generation, we add plans with all optimizations turned off and plans with all optimizations turned on, next to the randomly generated optimization plans. The value parameters for each of the initial compilation plans are set to the values used in the manually tuned default Jikes RVM configuration; this is the only aspect in which the framework relies on the experience of the compiler developer to start off with a meaningful set of values. Note that both these steps are not strictly required when using the framework; they do however significantly speed up and focus the exploration.

Table 6.3: The list of boolean optimizations which are available in Jikes RVM and used throughout this chapter. Optimizations used in the default Jikes RVM optimization levels `-O0`, `-O1` and `-O2` are indicated as such. Note that we also consider 15 optimizations which are not used in the default Jikes RVM optimization levels.

1	`-X:opt:local_constant_prop`	`-O0`
2	`-X:opt:local_copy_prop`	`-O0`
3	`-X:opt:local_cse`	`-O0`
4	`-X:opt:field_analysis`	`-O0`
5	`-X:opt:reorder_code`	`-O0`
6	`-X:opt:inline_new`	`-O0`
7	`-X:opt:inline`	`-O0`
8	`-X:opt:guarded_inline`	`-O0`
9	`-X:opt:guarded_inline_interface`	`-O0`
10	`-X:opt:preex_inline`	`-O0`
11	`-X:opt:monitor_removal`	`-O0,-O1`
12	`-X:opt:scalar_replace_aggregates`	`-O0,-O1`
13	`-X:opt:reorder_code_ph`	`-O0,-O1`
14	`-X:opt:inline_write_barrier`	`-O0,-O1`
15	`-X:opt:static_splitting`	`-O0,-O1`
16	`-X:opt:osr_guarded_inlining`	`-O0,-O1`
17	`-X:opt:osr_inline_policy`	`-O0,-O1`
18	`-X:opt:handler_liveness`	`-O0,-O1,-O2`
19	`-X:opt:redundant_branch_elimination`	
20	`-X:opt:ssa`	
21	`-X:opt:load_elimination`	
22	`-X:opt:coalesce_after_ssa`	
23	`-X:opt:expression_folding`	
24	`-X:opt:gcp`	
25	`-X:opt:gcse`	
26	`-X:opt:turn_whiles_into_untils`	
27	`-X:opt:global_bounds_check`	
28	`-X:opt:verbose_gcp`	
29	`-X:opt:licm_ignore_pei`	
30	`-X:opt:loop_versioning`	
31	`-X:opt:schedule_prepass`	
32	`-X:opt:reads_kill`	
33	`-X:opt:freq_focus_effort`	

The evolutionary search algorithm used in step 2 of the tuning process is configured in the same way, in terms of population and archive size, recombination operators and convergence criterion. The only difference is the entity definition, which describes a JIT compiler config-

Table 6.4: The list of value optimizations which are available in Jikes RVM and used throughout this chapter; the top 10 value optimizations steer one or more boolean optimizations, while the bottom 5 value options steer the AOS during adaptive compilation. For each value option, the value used in the default Jikes RVM compiler is shown.

-X:opt:ic_max_target_size	23
-X:opt:ic_max_inline_depth	5
-X:opt:ic_max_always_inline_target_size	11
-X:opt:ic_massive_method_size	2048
-X:opt:ai_max_target_size	119
-X:opt:ai_min_callsite_fraction_cole	0.40
-X:opt:unroll_log	2
-X:opt:cond_move_cutoff	5
-X:opt:load_elimination	3
-X:opt:infrequent_threshold	0.010
-X:aos:decay_frequency	100
-X:aos:dcg_decay_rate	1.10
-X:aos:dcg_sample_size	20
-X:aos:ai_seed_multiplier	3
-X:aos:ai_hot_callsite_threshold	0.0100

uration, i.e., a selection of optimization plans to be used at the different optimization levels and a set of value parameters representing the AOS settings, and the objective metrics, i.e., steady-state and startup performance relative to a baseline compiler. Using the 8 optimization plans retained from the first step, we construct all possible 92 entities for the initial population (subsequent generations only contain 25 entities): 8 entities which represent JIT compiler configurations with one single optimization level, and 28 and 56 entities which represent configurations with two and three levels, respectively. Again, the values of the AOS settings in the manually tuned default Jikes RVM configuration are used in the initial population.

Statistically rigorous performance evaluation

To deal with the non-determinism that is due to timer-based sampling and adaptive optimization in Jikes RVM, we use both multiple VM invocations and multiple benchmark iterations per VM invocation in our experiments, following the statistically rigorous performance evalua-

Table 6.5: Compilation rates and speedups over `base` on the Intel Core 2 for the optimization plans used by default in Jikes RVM (top rows), and the compilations plans obtained through our exploration (bottom rows).

Plan	Compilation rate	Speedup (over `base`)
`-O0`	53.12	1.86
`-O1`	21.84	2.14
`-O2`	20.81	2.13
A	59.70	1.77
B	57.62	1.86
C	50.86	1.89
D	41.07	2.00
E	37.42	2.02
F	28.70	2.05
G	25.90	2.08
H	19.11	2.13

tion methodology proposed by Georges et al. [43]. When reporting start-up performance we consider the average execution time for the first benchmark iteration across 20 VM invocations. When reporting steady-state performance we consider the arithmetic mean across the final 5 out of 15 benchmark iterations across 20 VM invocations. We report 95% confidence intervals which are indicated through error bars in the graphs.

We now evaluate the proposed JIT compiler tuning framework. We consider three cases: (i) tuning for average performance across all benchmarks, (ii) tuning for a particular benchmark, and (iii) tuning for a specific hardware platform. We consider experimental setups both with and without cross-validation. Finally, we discuss the exploration time.

6.3.2 Tuning for a benchmark suite

In a first evaluation, we use all the benchmarks from the SPECjvm98 and DaCapo suites, and aim at finding a JIT compiler configuration that performs well on average across all of the benchmarks. Our goal is to demonstrate that automated JIT compiler tuning performs at least as well as a manually tuned JIT compiler. This exploration was conducted on the Intel Core 2 platform.

Table 6.6: The JIT compiler configurations that are optimal in terms of startup (C_{ST}) and in terms of steady-state (C_{SS}).

	default	C_{ST}	C_{SS}
number of levels	3	3	2
level 0	P_{00}	plan A	plan A
level 1	P_{01}	plan C	plan E
level 2	P_{02}	plan E	–
Clock tick count for call graph decay	100	52	26
Call graph decay rate	1.10	1.10	1.10
Call graph update frequency (ticks)	20	3	4
Initial edge weight in call graph	3	3	3
Percentage of edges that mark hotness	0.01	0.0136	0.0098

Pareto optimal optimization plans

Table 6.5 lists the three default compilation levels as well as the optimization plans we obtained from the first step in our exploration process, in terms of compilation rate and speedup (code quality). The automatically derived Pareto optimal optimization plans are comparable to the manually tuned optimization plans in the manually tuned default Jikes RVM, and are well spread in terms of compilation rate and code quality.

Tuned JIT compiler

The second step is to identify optimum plan-to-level assignments and AOS settings. We denote the JIT compiler that yields best start-up performance as C_{ST}; the JIT compiler that delivers the best steady-state performance is denoted as C_{SS}. These settings are shown in Table 6.6. Interestingly, the optimum start-up JIT compiler C_{ST} has three levels with plans E, C and A, whereas the optimum steady-state JIT compiler C_{SS} has only two levels with plans E and A. We found the automatically tuned JIT compiler to achieve significantly better performance than the manually tuned Jikes RVM for a couple benchmarks, e.g., `mtrt` (30% for start-up and 7% for steady-state), `hsqldb` (10% for start-up) and `bloat` (3% for steady-state). For some benchmarks, we observe slightly worse performance, e.g., `lusearch` and `xalan` for steady-state; performance degradation is limited to 3% to 4% though. However, for the majority of the benchmarks, we do not observe statis-

tically significantly better or worse performance. Overall, the end conclusion is that automated JIT compiler tuning is feasible and achieves similar performance compared to a manually tuned JIT compiler.

Analysis

A number of interesting observations can be made from studying the selected Pareto optimal optimization plans and their composition, comparing them with the optimization plans of the default Jikes RVM optimization levels and looking into the optimization plans that are effectively used in the tuned JIT compilers.

A first observation is that the three optimization plans that deliver the highest performance (F-G-H) are not included in the best performing tuned JIT compilers obtained through our framework. Apparently the significant drop in compilation rate that accompanies these optimization plans, along with the very modest increase in delivered code quality makes these plans ill-suited for inclusion in the optimization level of an optimizing JIT compiler.

The optimization plan that is used in the highest optimization level of both the C_{SS} and C_{ST} JIT compilers, i.e., plan E, shows a compilation rate that is 80% higher than the optimization plan of the default `-O2` level, while yielding a speedup that is just 5.4% lower (see Table 6.5). This causes the adaptive controller to recompile hot methods at the highest optimization level a lot sooner in the tuned JIT compiler compared to the default configuration, causing an overall modest speedup for some benchmarks and a significant speedup in terms of startup performance for selected benchmarks, despite the slightly lower speedup delivered by the highest optimization level.

Figure 6.2 reveals the composition of the 8 optimization plans that were selected during the tuning process for both the SPECjvm98 and DaCapo benchmarks. Note that we excluded the value options, because the tuning framework did not change their values compared to the default values given to the plans in the initial set of plans.

A first striking observation regarding the composition of optimization plans is that the selection of enabled optimizations of a particular tuned optimization plan is not determined by the 'lower' optimization plans, as is the case with the optimization plans of the default Jikes RVM optimization levels. For example, in plan D various optimizations that were used in plan C are disabled, and other optimization were en-

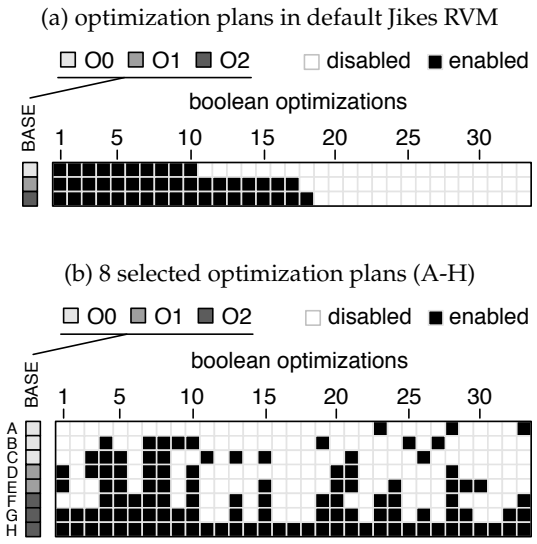


Figure 6.2: Composition of the optimization plans of the default Jikes RVM optimization levels (a), and the 8 selected optimization plans obtained with our framework, when tuning for both the SPECjvm98 and DaCapo benchmarks on the Intel Core 2 system. We excluded the value options for these optimization plans, because they remained unchanged by the tuning framework compared to the default values given during initialization of the exploration.

abled instead. This is a direct consequence of the automatic nature of the tuning framework: a compiler developer who needs to construct a set of optimization plans manually, trading of compilation rate and code quality, would be tempted to the more intuitive hierarchical way of constructing optimization plans, unknowingly paying the price in terms of performance in the end.

The most aggressive optimization plan used in the tuned JIT compilers, i.e., plan E, enabled 13 boolean optimizations. This explain the significant gain in compilation rate compared to the optimization plan of `-O2`, which enables 18 boolean optimizations. Plan E does not use 11 of the optimizations used in the `-O2` plan, but does enable 6 optimization not used at `-O2`, and achieves a better trade-off between compilation rate and delivered speedup that way. The selected optimization plan with the fastest compilation rate (plan A) does not use any of the optimizations used in `-O0` and only enables 3 optimizations all together. Plan C, which is used in the C_{ST} JIT compiler at the intermediate optimization level, enables half of the optimizations used at

`-O1` next to two other optimizations.

Several optimizations that are used in the optimization plans of the default Jikes RVM optimization levels are not used in the tuned optimization plans (except in plan H, which is shown to be of little interest because it represents a suboptimal trade-off): `local_copy_prop`, `scalar_replace_aggregates`, `inline_write_barrier`, `osr_guarded_inlining`, `osr_inline_policy` and `handler_liveness`. As mentioned in the previous chapter, there are several possible reasons for this: the optimizations are not applicable to the set of benchmarks used in the tuning process, they conflict with other optimizations, they hurt the compilation cost versus code quality trade-off, etc.

Of the optimizations that are not included in the default Jikes RVM v3.0.1 optimization levels, several are used in the tuned optimization plans. The most notable examples include `ssa`, `load_elimination` and `verbose_gcp`. While these optimizations might have appeared to be too expensive on top of the optimizations enabled at `-O2` to consider for inclusion, our results suggest that these optimizations might be worthwhile at various trade-off levels.

6.3.3 Tuning for a single benchmark

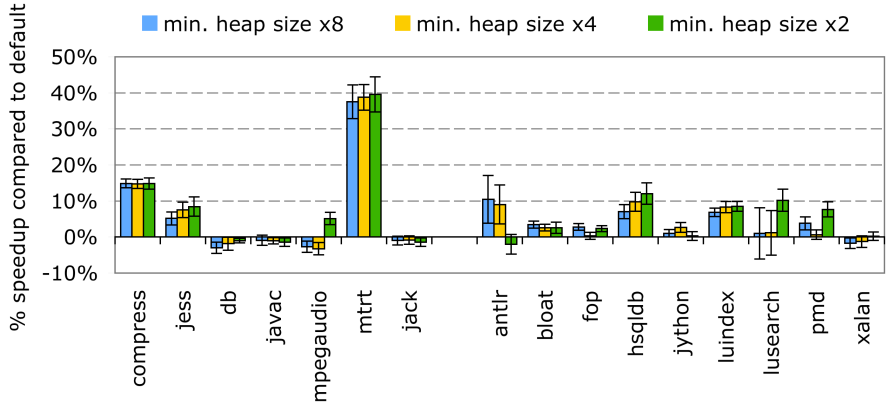
An important benefit from automated JIT compiler tuning is that it enables the optimization for specific applications as well as for specific hardware platforms at very low cost, given that the tuning process is completely automated. In this section, we discuss the results we obtain when we tune the JIT compiler for a specific benchmark; we discuss the case in which we tune for a specific hardware platform later.

Figure 6.3 shows the speedup on the Core 2 platform when comparing the best Pareto optimal configuration tuned per benchmark for (a) start-up and (b) steady-state performance against the manually tuned default Jikes RVM. The automated exploration yields JIT compilers that outperform the default Jikes RVM for a good portion of the benchmarks, and up to 40% for start-up and up to 19% for steady-state.

6.3.4 Cross-validation

The evaluation described so far assumed that the JIT compiler was tuned and evaluated using the same set of benchmarks, namely Da-

(a) start-up



(b) steady-state

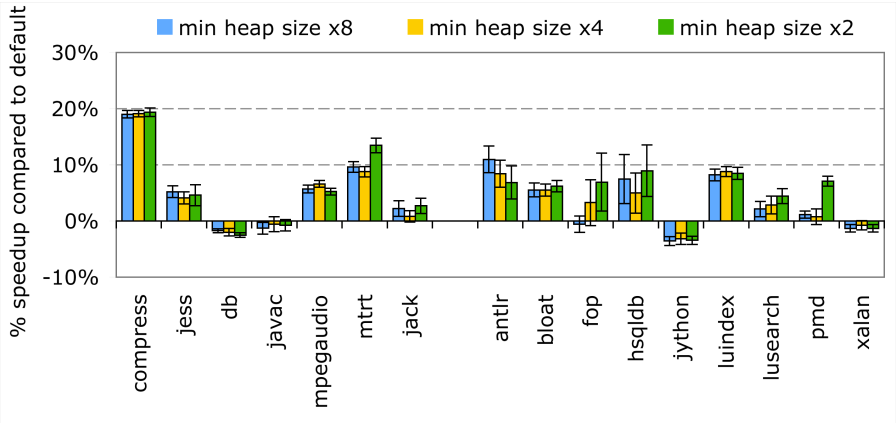
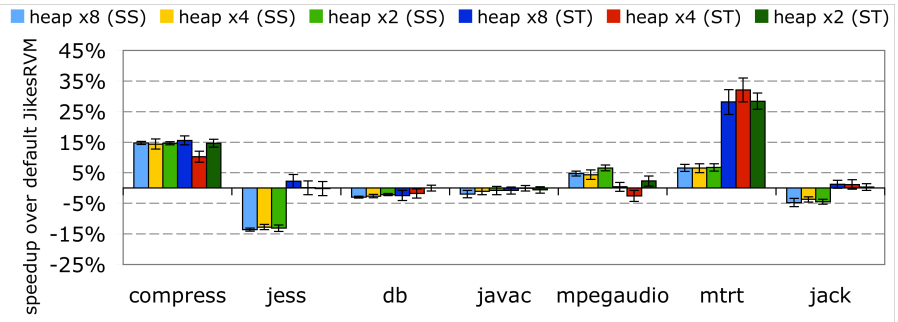


Figure 6.3: Speedup on the Intel Core 2 compared to the manually tuned default Jikes RVM for start-up and steady-state performance when tuning the JIT compiler for optimum performance on a per-benchmark basis.

(a) tuned for SPECjvm98, evaluated with SPECjvm98



(b) tuned for DaCapo, evaluated with SPECjvm98

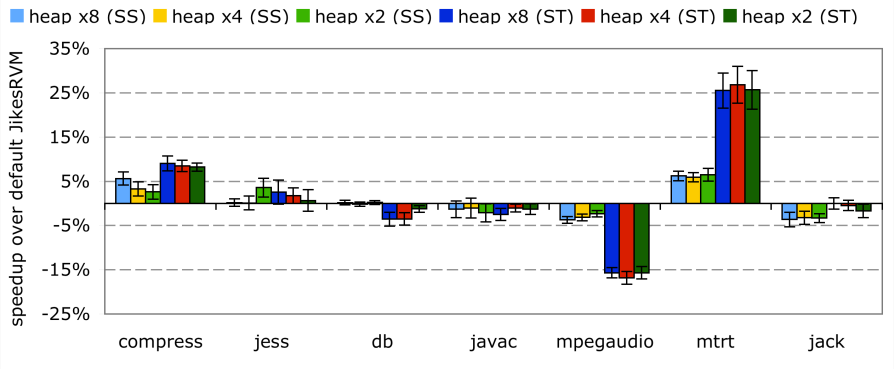


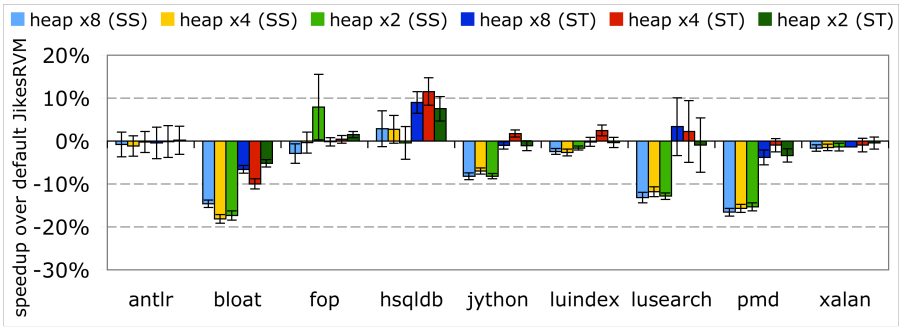
Figure 6.4: Per-benchmark performance speedups on the Intel Core 2 compared to default Jikes RVM when tuning Jikes RVM for the SPECjvm98 benchmarks in a non cross-validation setup (a) and a cross-validation setup (b). The graphs show results for both startup (ST) and steady-state (SS) performance, across three heap sizes.

Capo and SPECjvm98. Even more relevant is to study whether one could tune the JIT compiler with one set of benchmarks and then achieve good performance for other benchmarks. We now employ such a cross-validation setup: we tune the JIT compiler using the DaCapo benchmark suite and then evaluate the tuned JIT compiler using the SPECjvm98 benchmark suite, and vice versa. Figures 6.4 and 6.5 show the results of this cross-validation experiment along with the results of a non cross-validation experiment (i.e., the JIT compiler is tuned and evaluated using the same set of benchmarks), which serves as a point of reference.

For SPECjvm98, we observe that the automatically tuned JIT compiler achieves good performance even in a cross-validation experiment (compare Figure 6.4a to the non cross-validation experiment in Figure 6.4b). The automatically tuned JIT compiler achieves substantial speedups for mtrt and compress. We observe a slowdown for mpegaudio in the cross-validation setup. The performance picture is mixed for the DaCapo benchmark suite (Figure 6.5): when tuned for SPECjvm98, the JIT compiler performs worse for several DaCapo benchmarks, see for example bloat, jython, lusearch and pmd. For the other benchmarks, we observe similar (or similarly good, see hsqldb) performance under cross-validation. The reason for the different performance picture for DaCapo compared to SPECjvm98 is the significant differences in workload characteristics between DaCapo and SPECjvm98: Blackburn et al. [15] demonstrate that DaCapo shows more complex code, has richer object behaviors, and has more demanding memory system requirements. This result motivates the need for representative benchmarks when (automatically) tuning a JIT compiler—this is a general concern for feedback-loop based optimization and tuning.

In the previous section, we considered the same benchmark inputs when tuning the JIT compiler as during evaluation. Figure 6.6 reports performance results when considering a different input during the tuning process and evaluation, i.e., we now consider a cross-input validation setup. We limit ourselves to the DaCapo benchmarks in this experiment: we use the medium inputs during JIT compiler tuning and use the large inputs during evaluation. Two DaCapo benchmarks are excluded, namely `fop` and `luindex`, because the medium input is equal to the large input. We do not consider SPECjvm98 here because of lack of inputs: the `s1` and `s10` inputs are too small and only stress virtual machine startup performance and do not stress code quality [34].

(a) tuned for SPECjvm98, evaluated with DaCapo



(b) tuned for DaCapo, evaluated with DaCapo

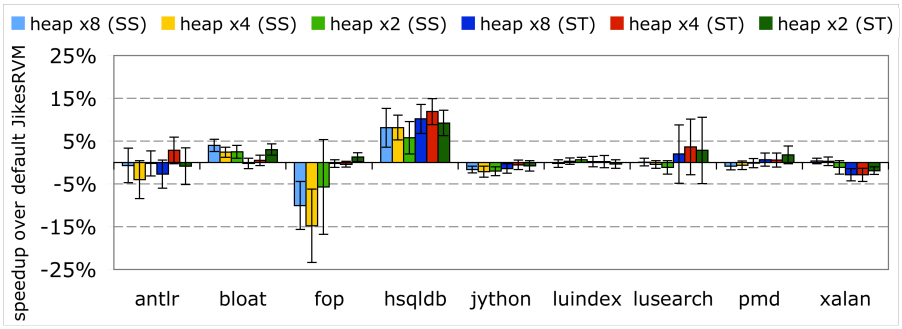
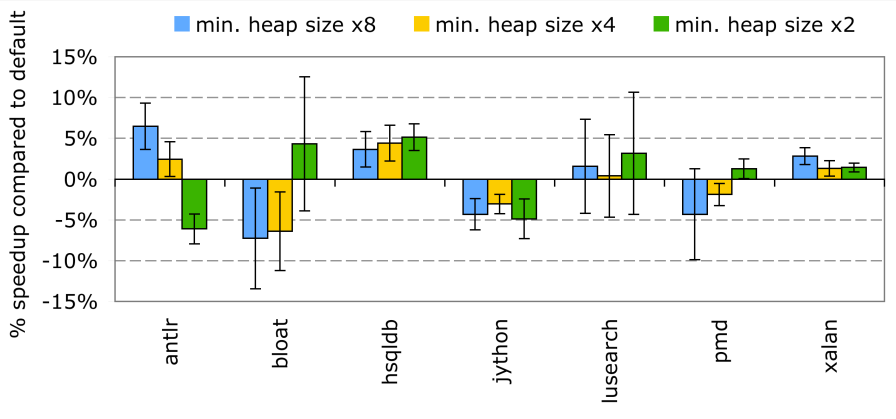


Figure 6.5: Per-benchmark performance speedups on the Intel Core 2 compared to default Jikes RVM when tuning Jikes RVM for the DaCapo benchmark suite in a non cross-validation setup (a) and a cross-validation setup (b). The graphs show results for both startup (ST) and steady-state (SS) performance, across three heap sizes.

(a) start-up



(b) steady-state

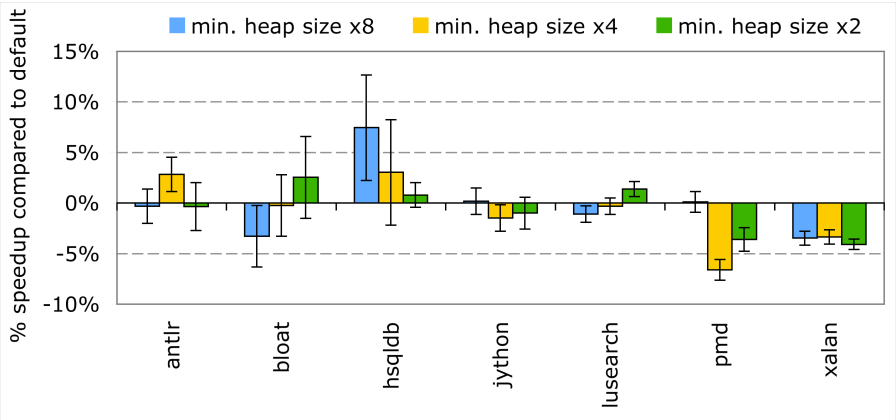


Figure 6.6: Per-benchmark start-up and steady-state speedup for a cross-input validation experiment.

Comparing Figures 6.3 and 6.6, we observe roughly the same speedup for the medium inputs (Figure 6.3) as for the large input (Figure 6.6) for some benchmarks, e.g., hsqldb. For other benchmarks, we observe a slight performance drop, e.g., bloat. This motivates the need for representative inputs when tuning a JIT compiler for a particular application — an input that yields substantially different program behavior than the input used during the tuning process may result in suboptimal performance. As mentioned before, this is a general concern for feedback-loop based optimization and tuning.

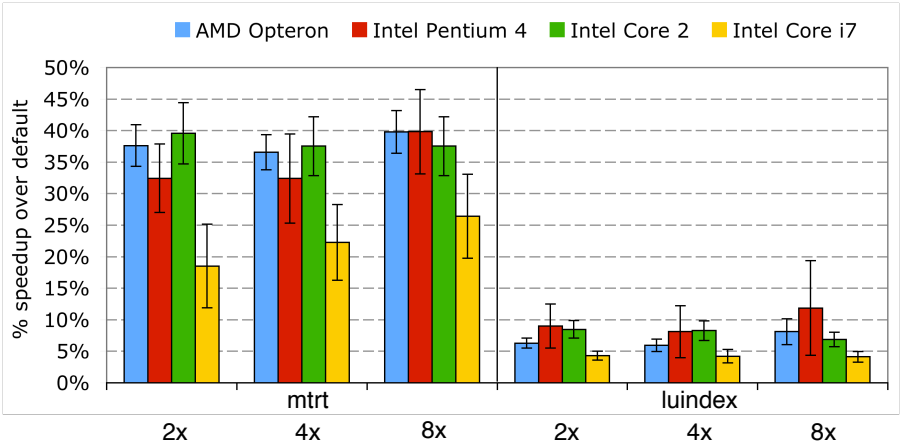
6.3.5 Tuning for a specific hardware platform

We now explore the potential performance benefit by tuning the JIT compiler for a specific hardware platform. In this section, we examine the effects of tuning for a particular platform using two benchmarks: (i) `mtrt`, and (ii) `luindex`. During the benchmark suite wide exploration on the Intel Core 2 (see Figures 6.4 and 6.5), the optimum JIT compiler did very well for `mtrt`, yet it failed to improve performance for `luindex`. In the per-benchmark exploration, `mtrt` improves further, while `luindex` gains 9% in both start-up performance and in steady-state performance on the Core 2, see Figure 6.3. Thus, these two benchmarks make for two excellent cases for examining the effect of exploring benchmark-specific tuning across different hardware platforms.

Per-platform tuning

Our first experiment tunes the JIT compiler for a specific hardware platform and compares performance against the default JIT compiler (which was manually tuned for another hardware platform). The results of this hardware-specific exploration are shown in Figure 6.7 for (a) start-up and (b) steady-state performance. There is significant benefit in start-up performance for `mtrt`: performance speedups range from 19% to 40%. For `luindex` we observe performance benefits in the 4% to 12% range. For steady-state performance, we observe substantial performance benefits for both `mtrt` and `luindex`: performance improves by up to 13% for `mtrt` and up to 17% for `luindex`. These examples make the case that significant speedups can be obtained from tuning a JIT compiler for a specific hardware platform.

(a) start-up



(b) steady-state

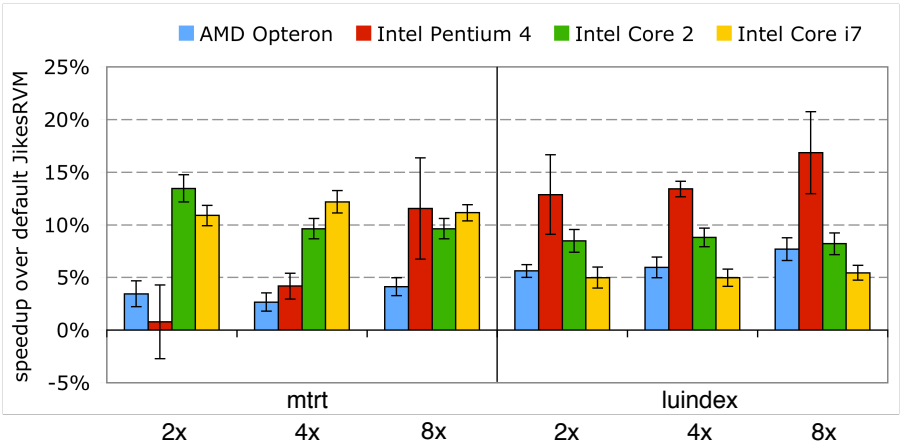


Figure 6.7: Speedup numbers across different heap sizes on all hardware platforms for `mtrt` and `luindex` for (a) start-up and (b) steady-state performance.

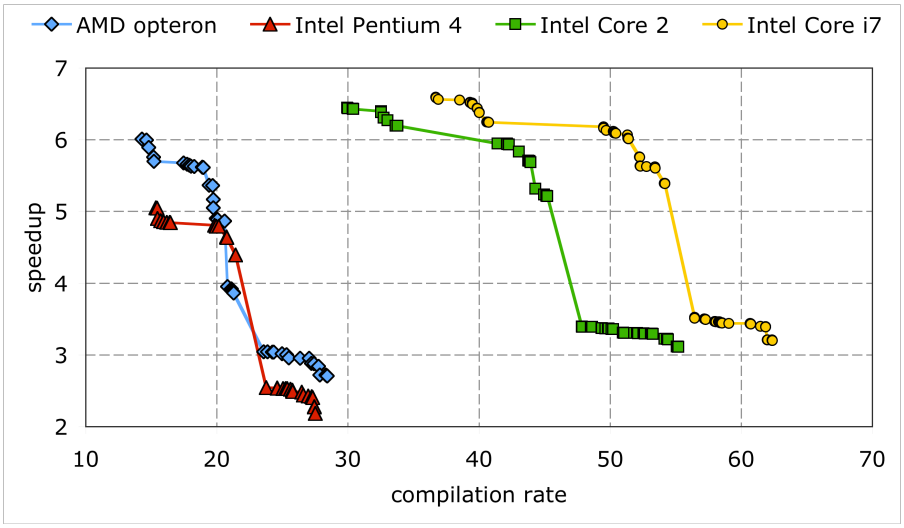


Figure 6.8: The Pareto frontiers for the optimization plans tuned for `mtrt` on each of the platforms in our experimental setup.

This result is further illustrated in Figure 6.8 which shows the per-platform Pareto optimal optimization plans in terms of compilation rate and performance speedup relative to baseline compiled code. There are two observations we can immediately draw from this graph. First, the frontier shifts to the upper right for more recent platforms. As a consequence, if one tunes the compiler DNA on an older platform, the cost for optimization is over-estimated and the potential benefit the optimization reaps is under-estimated. This may result in optimizing later — more samples are required to reach the decision threshold — or optimizing to a lower level. Conversely, if the VM is tuned for a more recent platform, the VM might optimize too soon and/or too much, potentially offsetting the gain the adaptive framework might bring. Second, we see that on each platform the Pareto frontier is well spread across the space, suggesting that a few optimizations might have a large effect.

Employing tuned JIT compilers across platforms

Using a JIT compiler that was tuned for a particular hardware platform on another hardware platform may yield suboptimal results. This is illustrated in Figure 6.9a where the Pareto optimal plans tuned for `mtrt`

for each platform have been evaluated on the Intel Core 2 platform. The optimization plans tuned for the Intel Pentium 4 and AMD Opteron platforms are suboptimal, i.e., they perform worse than the ones that were tuned for the Core 2. We observe a similar result when looking into tuned JIT compilers, see Figure 6.9b which compares the performance of a JIT compiler tuned for the Pentium 4, AMD Opteron and Core i7 when run on a Core 2 machine against a JIT compiler that was tuned on the Core 2. In this figure, we only show the two JIT compiler configurations that perform best in terms of steady-state and startup. Clearly, the JIT compilers tuned for the Core 2 yields the best possible performance on the Core 2—the JIT compilers tuned for the other platforms perform worse. In particular, the JIT compiler that was tuned for the Core 2 yields approximately 5% better start-up performance and 10% better steady-state performance compared to the JIT compiler that was tuned for the Pentium 4. We thus conclude that platform-specific JIT compiler tuning can yield substantial performance benefits and transferring JIT settings across platforms may lead to suboptimal performance.

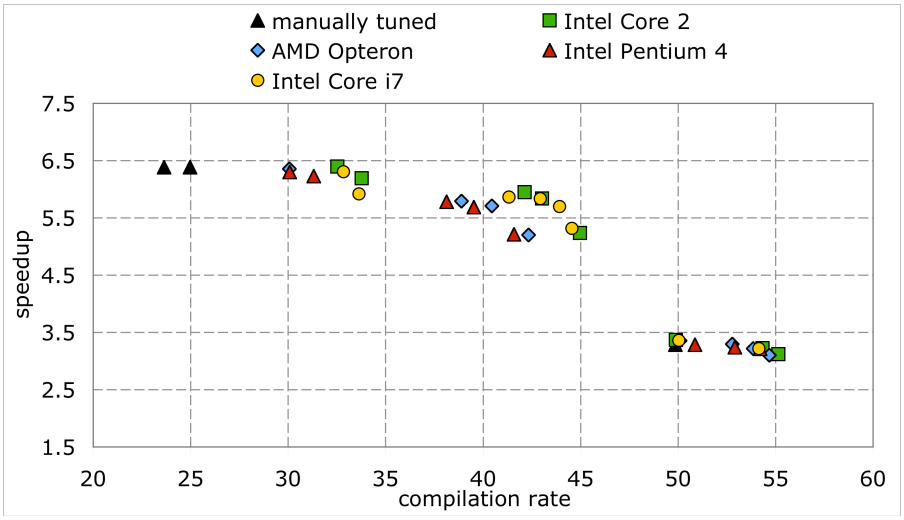
6.4 Exploration time

Finally, we discuss how much time is needed to complete the JIT compiler tuning.

Performing the first step for all of the DaCapo and SPECjvm98 benchmarks on the Core2 platform took 33 generations to converge. During each generation, 25 new optimization plans are constructed, each of which requires roughly 40 minutes to measure the compiler DNA. This means that about 550 machine hours are needed to run the first step of the tuning process to convergence. Note however, that this tuning process is again embarrassingly parallel, i.e., all plans can be measured in parallel and independently of each other. Thus, having sufficient machine resources available this first step only takes about 22 hours.

The second exploration step, which tunes the plan-to-level assignment and AOS, converges significantly faster: only 8 generations are required. Evaluating a single JIT compiler setting takes about 400 minutes on average. This results in an additional exploration time of about 1320 hours. Again, because each generation can be evaluated in parallel, the exploration can be performed in about 53 hours.

(a) Optimization plans



(b) Tuned JIT compilers

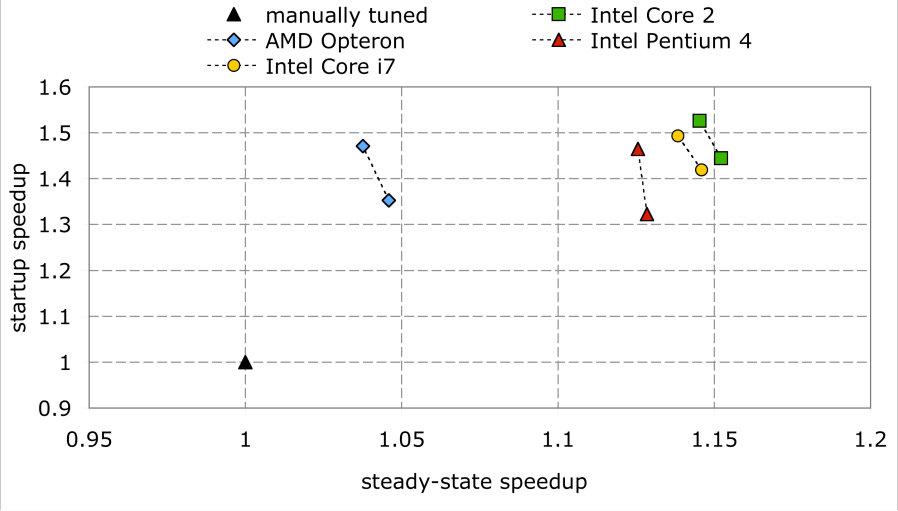


Figure 6.9: Graph (a) shows compilation rate versus performance speedup for the Pareto optimal optimization plans determined on the AMD Opteron, Pentium 4 and Core i7 when run on the Core 2 platform. Graph (b) shows start-up versus steady-state performance for the AMD Opteron, Pentium 4 and Core i7 tuned JIT compilers when evaluated on the Core 2. These graphs consider SPECjvm98's `mtrt`.

Thus, the entire exploration for a set of 16 benchmarks on a particular hardware platform takes around 75 hours or roughly 3 days. Performing the exploration for a single application only, as we did for the experiments described in Section 6.3.3, is a matter of hours.

6.5 Related work

Most modern Java virtual machines implement multiple levels of optimization, see for example [7, 9, 76, 97]. Since compilation time is an integral part of the total execution time in a dynamic compiler, it is of utmost importance to make a good trade-off between compilation time and code quality when proposing optimization levels in a optimizing dynamic compiler. Arnold et al. [9] and Ishizaki et al. [62] describe how optimization levels are determined manually for the Jikes RVM and IBM DB production VM, respectively.

Cavazos and O'Boyle [21] take a different approach to optimizing JIT compilers. They apply a different optimization plan for each method. The optimizations in these plans are determined by using a logistic regression function that predicts which optimizations are most useful for the given method based on bytecode features. They report speedups of 4%, 2% and 29% on average compared to optimizing all methods at the `-O0`, `-O1`, and `-O2` levels, respectively. When considering adaptive optimization, they report a 1% improvement over default Jikes RVM for SPECjvm98; for the DaCapo benchmarks, they report a 4% average performance improvement. Our JIT tuning approach does not apply different optimization plans to individual methods which simplifies JIT compiler tuning. In addition, Cavazos and O'Boyle do not make the case that JIT compilers that are tuned for particular hardware platforms and/or applications can yield substantial performance benefits. In their follow-on work [20], Cavazos and O'Boyle use a genetic algorithm to automatically tune the heuristics of the inliner in a dynamic Java compiler. Our work is not limited to the inliner; we instead tune the entire JIT optimization system.

6.6 Summary

In this chapter, we presented a framework for automatically tuning dynamic compilers. The framework uses evolutionary searching and tunes the JIT compiler for a given hardware platform and a given application or application domain [57]. This is done through a two-step process in order to manage the complexity in exploring the huge optimization space: we first identify Pareto optimal optimization plans, and subsequently assign plans to optimization levels and fine-tune the AOS.

Our experimental results using the Jikes RVM, four hardware platforms and the SPECjvm98 and DaCapo benchmarks, demonstrate that the proposed framework identifies JIT compiler configurations that achieve significantly better performance compared to a manually tuned VM. When optimizing for individual applications, we achieve performance improvements up to 40% and 19% for start-up and steady-state performance, respectively. Also, optimizing for a specific hardware platform leads to significantly better performance. Our framework is completely automated and explores the complex JIT compiler space in approximately 3 days for the collection of DaCapo and SPECjvm benchmarks; tuning the JIT compiler for individual applications is done in a few hours.

Chapter 7

Conclusions and Future Work

One never notices what has been done;
one can only see what remains to be done.
Marie Curie

7.1 Conclusions

Modern computer systems are utterly complex pieces of engineering. The most recent microprocessors are implemented using billions of transistors, and consist of a multitude of components working together and delivering great performance compared to previous generations. Although the design of increasingly better performing computer systems is kept manageable by introducing several layers of abstraction, this complexity is still a concern for computer engineers. In this dissertation we looked into a number of problems related to the performance of modern computer systems, for which the cause can be traced back to their complexity.

7.1.1 Analyzing and estimating performance

A first set of problems we studied relate to the analysis and estimation of computer system performance.

Analyzing inherent time-varying program behavior

In Chapter 2 we presented a set of microarchitecture-independent workload characteristics that allow for capturing the true inherent behavior of applications, as opposed to hardware performance counter metrics which are commonly used in workload characterization studies. Based on these workload characteristics, we presented a methodology for studying the phase-level behavior of a set of workloads, which is more informative than the less detailed aggregate workload characterization as mostly done in current practice. The methodology combines different machine learning techniques including k-means clustering, Principal Component Analysis (PCA) and genetic algorithms to cope with the large amount of information that needs to be processed, and to distill the most relevant information. Subsequently, we determine the most prominent phase behaviors, identify the key microarchitecture-independent workload characteristics and visualize these prominent phase behaviors using kiviat diagrams. Applying the methodology on a data set consisting of over one million execution intervals obtained for 77 benchmarks demonstrated the applicability of our methodology, and yielded interesting insights when comparing the workload behavior of the different applications and benchmark suites. Additional analysis in terms of the coverage, diversity and uniqueness of the benchmark suites confirmed several intuitive understandings, e.g., that domain-specific benchmark suites represent a much smaller part of the workload space than general-purpose benchmark suites like SPEC CPU, and that general-purpose benchmark suites exhibit less unique behavior than benchmark suites like BioPerf and BioMetricsWorkload, which represent emerging workload domains.

Recognizing and interpreting performance trends

We proposed an analysis framework in Chapter 3 named Processor Performance Visualizer (PPV), which also uses PCA for studying performance data across a large number of computer systems. By using PCA as a feature extraction technique, we are able to quickly and easily recognize performance trends in a large data set of performance numbers. By relying on the microarchitecture-independent workload characteristics, we were able to give a meaningful interpretation to each of the principal components representing underlying dimensions in the data set. Thus, we were able to give meaning to each of the recognized per-

formance trends, allowing for obtaining interesting insights. We applied the PPV methodology to the data sets of performance numbers available for both the SPEC CPU2000 and SPEC CPU2006 benchmark suites. This revealed performance trends over a wide range of computer systems, such as the difference between the speed demon and brainiac computer systems, the relation between increased clock frequency and the performance for memory-intensive workloads, notable differences between microprocessors of different vendors and the various sub-generations in certain processor families.

Estimating relative system performance

Next to analyzing inherent program behavior and looking for performance trends, we also developed a methodology to anticipate how good computer systems perform for a particular application-of-interest, relative to each other. While common practice often relies on using the average performance across a range of benchmarks for ranking systems, we estimate the performance based on the performance numbers obtained for benchmarks exhibiting program behavior that is similar to that of the application-of-interest. Using a genetic algorithm, we first learn the relation between the difference in terms of microarchitecture-independent workload characteristics of each benchmark pair and the corresponding difference in terms of performance, and then weight each of the workload characteristics accordingly. To find the most similarly behaving benchmarks for a particular application, we use the k-nearest-neighbors technique. The performance of the application-of-interest is then computed as the weighted average of the performance numbers of the proxy benchmarks. Evaluating this methodology using the SPEC CPU2000 and CPU2006 benchmarks using different sets of computer systems showed significant improvements in terms of overall ranking over current practice which relies on overall average performance. For large sets of systems, i.e., sets counting over 1,000 systems, our methodology showed improvements in terms of the average Spearman rank correlation coefficient from 0.897 to 0.923 and from 0.857 to 0.874 for CPU2000 and CPU2006, respectively. For a small number of computer systems considered, we saw an increase in the average Spearman rank correlation coefficient from 0.833 to 0.892 and from 0.696 to 0.731 for CPU2000 and CPU2006, respectively. Some outlier benchmarks result in low Spearman correlation coefficients, both for current practice and our estimation frame-

work, because they exhibit program behavior that is significantly different from all of the other benchmarks in the suite.

7.1.2 Automatically specializing system software

In Chapters 5 and 6, we looked at the problems that arise when optimizing applications for a particular hardware platform.

Constructing optimization levels for a static compiler

For static compilers a set of manually constructed optimization levels is typically provided, with each optimization level representing a different trade-off between objectives such as compilation time, code size and code quality. Constructing these optimization levels is tedious and time-consuming for a variety of reasons. An in-depth knowledge of the multitude of possibly interacting compiler optimizations and the dependency of their effect on both the application being compiled and the target platform is required. To help resolve this issue, we proposed the COLE framework which allows for constructing compiler optimization levels fully automatically, while trading off between a number of objectives. The experimental evaluation we presented showed that: (a) our framework based on a multi-objective evolutionary search algorithm significantly outperforms random searching, (b) the obtained optimization levels represent better trade-offs than the manually constructed default optimization levels, (c) specializing the optimization levels to a particular hardware platform or application domain is important to provide good trade-offs. Thorough analysis of the composition of the optimization levels obtained through COLE revealed interesting insights concerning the usefulness of individual compiler optimizations, depending on the target platform and the trade-off being made.

Automatically tuning a Just-In-Time compiler

Building on this work, we also presented an automated tuning framework for Just-In-Time compilers. Specializing the optimizing recompilation mechanism of modern JIT compilers, which relies on a number of different optimization levels and an adaptive controller to steer the recompilation, was found to be significantly more difficult than constructing optimization levels for a static compiler. On top of the already

challenging task of finding optimization plans that represent suitable trade-offs in terms of compilation cost and code quality, a selection needs to be made of plans that perform well together in an adaptive optimization setting, and the adaptive controller needs to be fine-tuned. The framework we proposed in Chapter 6 uses a two-step approach to resolve these issues. In a first step, a set of Pareto-optimal optimization plans is obtained, from which a selection of plans representing different trade-offs between (re)compilation cost and code quality is retained. Using these optimization plans, JIT compilers utilizing different optimization levels are constructed and fine-tuned. Using the Jikes RVM and a collection of Java benchmarks, our experimental results show that the framework is able to deliver JIT compilers that perform on average equally well as JIT compilers for which the optimization plans and parameters of the adaptive controller were obtained manually. Furthermore, we showed that using the framework to automatically specialize the JIT compiler to a particular application and a particular hardware platform, a task previously unthinkable because of the labor-intensive re-tuning involved, results in speedups of up to 40% for startup performance and 19% for steady-state performance. Empirical results were given to show that re-tuning the JIT compiler for a different target platform yields significant performance improvements.

7.1.3 Efficacy of machine learning techniques

Through the various solutions we proposed, we showed that resorting to machine learning techniques allows for solving difficult problems related to computer system performance, each of which is tightly coupled to the sheer complexity of modern-day computer systems. By picking machine learning techniques that are well-suited for the particular problems at hand, e.g., PCA for extracting underlying dimensions in large data sets and evolutionary search algorithms for finding adequate solutions in huge search spaces, the proposed methodologies yield good results overall. Resorting to these techniques relieves computer engineers from tedious time-consuming tasks by automating them, and often yields better results because they do not simply depend on high-level heuristics and intuition.

7.2 Future work

Several of the methodologies proposed in this dissertation are subject to improvement, in order to make them even more applicable in real-world situations. This section briefly describes some interesting possibilities for future work.

Recently chip-multiprocessors, often referred to as multi-core processors, have become the industry standard for general-purpose computing. As mentioned before, important reasons for this paradigm shift are the diminishing returns in performance improvements and challenging power requirements that accompany additional microarchitectural improvements targeted to single-thread performance. One interesting extension to the set of microarchitecture-independent workloads characteristics presented in Section 2.1.3 would be to add support for multi-threaded workloads. In particular, workload characteristics that provide insight into the inter-thread communication patterns and potential resource conflict behavior between threads would be valuable.

The performance estimation framework presented in Chapter 4 could be further improved to obtain more accurate performance estimations. Being able to obtain accurate estimations of the time-varying behavior of applications would contribute to the applicability of the methodology. Another interesting addition would be the ability to detect outliers a priori, for which performance estimation based on the inherent behavior of the available set of benchmarks is unlikely to be accurate, and/or to quantify the level of confidence in the estimated performance numbers.

An important limiting factor of the automated frameworks presented in Chapters 5 and 6 is the time required to construct well-performing optimization levels or JIT compilers. Although the exploration time needed by the current frameworks is very small relative to the time that would be needed to evaluate each point in the respective design spaces, this still may be troublesome in a commercial setting where time-to-market is of critical importance. To further limit the exploration time, the evolutionary search algorithm could be combined with predictive models for each of the objectives, which would eliminate the need for measuring each of the objective functions for every candidate solution being considered. Related machine learning techniques such as active learning and global optimization have shown promising results in different research domains.

Appendix A

Machine Learning Techniques

A.1 Principal Component Analysis

Principal Component Analysis (PCA) [64] is a data analysis technique with two interesting features: (i) it transforms a given data set with (potentially) highly correlated dimensions into a data set with uncorrelated dimensions, and (ii) it allows lowering the dimensionality of the data set while controlling the amount of information that is lost. These features are important, for two reasons. First, analyzing a lower dimensional space is easier than analyzing a higher dimensional space. Second, analyzing correlated data gives a distorted view. By removing correlation from the data set, equal weight is given to all underlying dimensions represented by the principal components.

The input to PCA is a matrix in which the rows are the *instances* and the columns are the *variables*. PCA computes new variables, so called principal components, which are linear combinations of the original variables. This is done such that all principal components are uncorrelated. More formally, PCA transforms the p variables $X_1, X_2, \ldots, X_p$ into p principal components $Z_1, Z_2, \ldots, Z_p$ with $Z_i = \sum_{j=1}^{p} a_{ij} X_j$. The a_{ij} coefficients are computed by PCA and are the factor loadings of variable X_j for principal component Z_i.

This transformation has the following properties:

(i) the principal components are ordered by decreasing variance:

$$Var[Z_1] \geq Var[Z_2] \geq \ldots \geq Var[Z_p]$$

i.e. Z_1 contains the most information and Z_p the least;

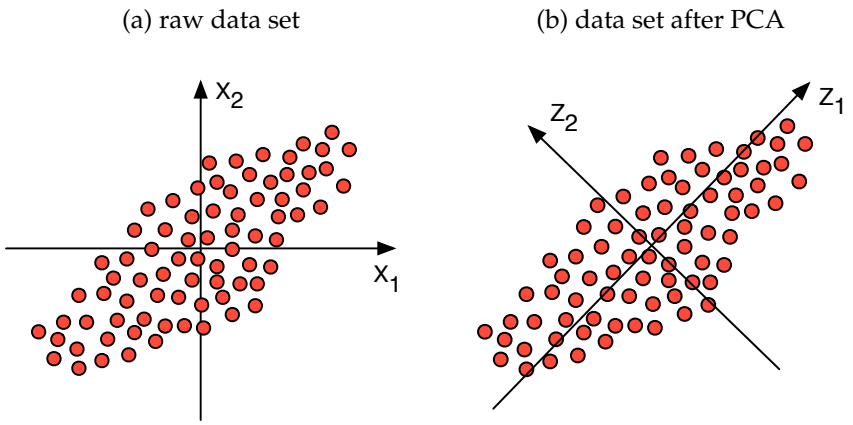


Figure A.1: Principal Component Analysis on a hypothetical 2D data set; (a) the data in terms of the original variables X_1 and X_2, (b) the transformed data in terms of principal components Z_1 and Z_2.

(ii) there is no covariance between principal components:

$$Cov[Z_i, Z_j] = 0, \forall i \neq j$$

i.e. there is no information overlap between the principal components, or, in other words, they are uncorrelated;

Note that the total variance in the data (variables) remains the same before and after the transformation, i.e. $\sum_{i=1}^{p} Var[X_i] = \sum_{i=1}^{p} Var[Z_i]$.

An illustrative example of how the principal components relate to the original input dimensions in shown in Figure A.1 for a hypothetical 2-dimensional data set.

A.1.1 Normalization

Before applying PCA, it is common to first normalize the data set. This is done by subtracting the mean $\bar{x}$ and dividing by the standard deviation s for each input variable X_i $(1 \leq i \leq p)$: $X_i^{'} = \frac{X_i - \bar{x}}{s}$. This causes the transformed variables $X_i^{'}$ to have a zero mean an unit standard deviation. This normalization step is illustrated in Figure A.2.

The motivation behind normalizing prior to PCA is that the data set is often heterogeneous in terms of its input variables, and thus the variance of the original input variables may differ substantially. Considering the microarchitecture-independent workload characteristics used

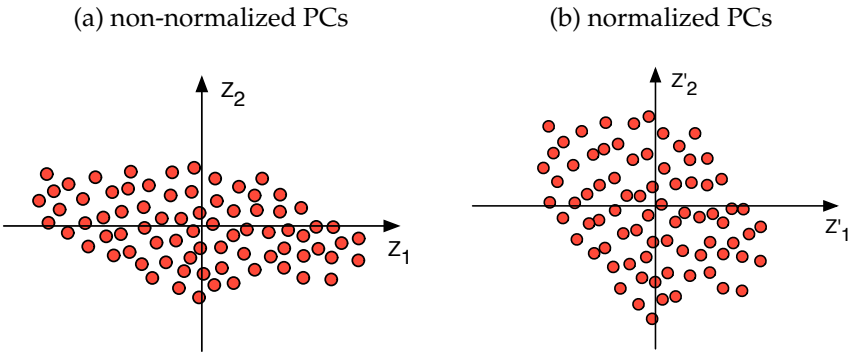


Figure A.2: Normalizing principal components using a hypothetical 2D data set; (a) the raw (non-normalized) principal components Z_1 and Z_2, (b) the normalized principal components Z'_1 and Z'_2.

in this dissertation, this is indeed the case; e.g., the variance of the amount of ILP is likely to be a lot higher than the variance of the ratio of integer instructions. With respect to PCA, normalizing the data set is crucial because in the case of non-normalized data a higher weight would be given in the analysis to variables with a higher variance.

After applying PCA it is also common to normalize the resulting principal components, depending on the context they will be used in. The idea is that PCA finds the key underlying mechanisms that correlate well with the data set. By normalizing the principal components, these underlying mechanisms get equal weights in the subsequent analysis.

A.1.2 Reducing the dimensionality

By removing the principal components with the lowest variance, we can reduce the dimensionality of the data set while controlling the amount of information lost. We retain the first q of p principal components which results in a significant reduction in dimensionality, since $q \ll p$ in most cases. Determining the number of principal components to retain is an important issue: too few principal components will fail to capture the major trends in the data set, while too many principal components complicate reasoning about the data set and may lead to a curse-of-dimensionality problem. To measure the fraction of information retained in this q-dimensional space, we use the amount of

variance accounted for by these q principal components:

$$\frac{\sum_{i=1}^{q} Var[Z_i]}{\sum_{i=1}^{p} Var[Z_i]}$$

For example, criteria such as '80% or 90% of the total variance should be explained by the retained principal components' is often used for dimensionality reduction. Alternative criteria are to retain all principal components that explain a fraction of the total variance that is at least as large as the minimum variance of the original variables, or to retain all principal components with a standard deviation greater than one.

A.1.3 Interpretation of principal components

It is often argued that principal components can be easily interpreted, because they are basically linear combinations of the original variables. Arguably, a coefficient a_{ij} that is very high or very low compared to other coefficients implies a strong impact of the original variable X_j on the principal component Z_i. On the other hand, a coefficient a_{ij} that is close to zero implies very little impact.

In practice however, interpreting principal components sometimes turns out to be surprisingly hard. For a principal component with both positive and negative factor loadings, positive contributions by certain variables balance out the negative contributions by others, which might complicate giving a sensible interpretation to each of the principal components. This is illustrated in Chapter 2 of this dissertation.

A.2 Genetic algorithms

A *genetic algorithm (GA)* is an iterative search algorithm which tries to improve intermediate solutions by tweaking them or combining them to obtain new, hopefully better solutions. Genetic algorithms are a well-known and effective machine learning technique, based on the biological process of evolution. They have proven to be very effective in large search or optimization spaces, for a wide variety of problems including multiple sequence alignment (bioinformatics) [46], control engineering [72] and numerical optimization [74].

In this work, we rely on a number of different types of so-called *evolutionary algorithms (EAs)* to tackle a variety of problems. Although

strictly speaking genetic algorithms are a special case of evolutionary algorithms, we will use both terms interchangeably. For an overview of evolutionary algorithms in general and genetic algorithms specifically, we refer to the work of Thomas Bäck [11]. In this section, we will give an overview of genetic algorithms in general and detail on the way in which we used them to identify key workload characteristics.

Although there is a general definition of a genetic algorithm which is followed by many, it is often the subtleties of a particular implementation and choice of parameters that make the difference between impressive or downright disappointing results [31, 101]. In the following paragraphs, we will detail on the particular implementation of genetic algorithms we used in our experiments. We describe the terminology used in subsequent sections and discuss the aspects of genetic algorithms that we found to be most important to obtain good results for a variety of problems.

A.2.1 Terminology

A genetic algorithm applies the mechanism of evolution to one or more *population(s)* of *entities*, together forming a *generation*, and keeps track of the best entities so far in the *archive*.

An entity is the equivalent of a biological entity, i.e. a useful representation of a (candidate) solution for the problem at hand. Examples range from a single vector of elements (e.g. bits, integer numbers, *etc.*) to a combination of multiple parameters, vectors and other structures containing the required information to fully describe the candidate solution. The different parts of an entity are often referred to using terms like chromosomes, genes and alleles, depending on the granularity. We will try and make the description of the entities and the operators working on them more explicit to avoid these latter, sometimes confusing terminology.

A population is simply a collection of entities in one iteration of the GA. The set of populations which is being used in one particular iteration of the GA is called the *generation* of (populations of) entities. Thus, a genetic algorithm iterates across a number of generations, gradually improving the best entities. Between generations, the set of best entities so far is referred to as the *archive*, and will be important both for generating new entities in subsequent generations and for detecting convergence.

A.2.2 Defining entities

Using a well-suited entity definition, which is very often specific to the problem of interest, is an important part of configuring the genetic algorithm. Indeed, all the information needed to model all the necessary aspects of candidate solutions in order to quantify their quality should be present. On the other hand, a complex definition significantly complicates defining suitable entity operators, which in turn results in obtaining suboptimal or downright disappointing solutions for the problem (see Section 6.2.1 in this dissertation, for example).

In most cases where we will rely on genetic algorithms to find good solutions, we will limit ourselves to simple entity definitions, such as vectors of atomic elements like bits, integer and/or floating-point numbers. This will prove sufficient in order to obtain satisfying results relatively quick, and more importantly will also allow us to use more or less standard and well-established crossover and mutation operators.

A.2.3 Crossover and mutation operators

There are two important entity operators which are used in the construction of new entities in subsequent generations: *crossover* and *mutation*.

The crossover (or recombination) operator is applied to two entities, and usually results in two new entities. In this, the objective is to try and combine the good features of both parent entities and produce offspring entities of higher quality. Usually, the operator has at least one control parameter which can be tweaked to steer the amount of information that is exchanged between the parent entities.

Figure A.3 illustrates two well-known examples of crossover operators: *crossover mixing*, in which multiple parts of an entity are exchanged randomly in isolation (e.g. vector elements), and n*-point crossover*, in which n so-called crossover points are chosen at random after which contiguous parts of the entity definition are exchanged (e.g. parts of a vector). The control parameters for these particular crossover operators are the probability of exchanging information between entities, and the number of crossover points n, respectively.

Next to crossover, the mutation operator also plays an important role in the construction of new entities. This operator is applied to a single entity, and randomly changes or tweaks one of multiple parts

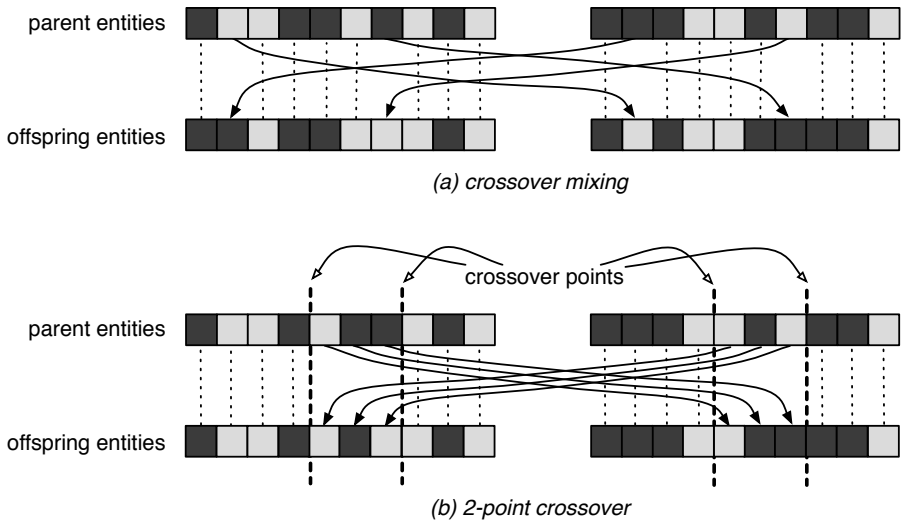


Figure A.3: Application of the crossover mixing and 2-point crossover operators on two entities, each represented by a simple bit vector.

of the entity. The idea is that these random changes hopefully yield an increase of the quality of the entity. While the crossover operator only combines features of entities already evaluated, the mutation operator allows to generate entities with features which were not used before. Thus, it helps to avoid getting stuck in local optima in the search space. Just like crossover, the mutation operator often is steered by one or more parameters. Usually, one parameter controls the aggressiveness with which entities are mutated, i.e. how many parts of the entity are randomly changed or tweaked.

To illustrate the way in which mutation works, we consider an example of a frequently used mutation operator: *multi-point drift*, illustrated in Figure A.4. Multi-point drift will tweak multiple random parts of the entity, taking into account a control parameter that expresses the degree in which the mutated part should differ from the original part. In the illustrative example in Figure A.4 we use a vector of values represented by different shades of gray. Each vector element is mutated with a probability of 0.2, and a large value for the parameter controlling the degree of mutation is used. Thus, if a vector element is mutated, a rather large difference between the original and mutated value is observed.

The larger part of a new generation of entities is usually created

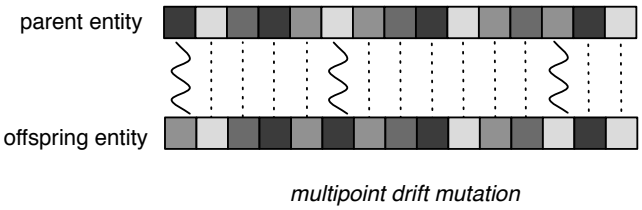


Figure A.4: Application of the multi-point drift mutation operator on an entity defined as a vector of values. Each vector element is represented by a shade of gray, for simplicity.

using the crossover operator, while the rest is generated using the mutation operator. The motivation behind this is to let the inheritance of entity features to the offspring entities dominate, and thus to limit the amount of totally random changes. As the ratio of mutated entities to recombined entities increases, the genetic algorithm tends to lean towards random search. The main purpose of the mutation operator is to supply entities that use a large range of values to the crossover operators [11]. Therefore, we will usually use crossover and mutation rates in the range of 80-90% and 10-20%, respectively.

A.2.4 Fitness, selection, evolution and convergence

As should be clear from the previous paragraphs, an important aspect in supporting the genetic algorithm to gradually evolve towards increasingly better entities is the ability to recognize high quality entities. The quality of an entity is computed by a so-called *fitness function*, of which the exact definition is very dependent on the entity definition and the particular problem at hand. We will use the term *fitness score*, or *fitness* for short, to denote the quality of an entity as computed by the fitness function.

During the exploration conducted by the genetic algorithm, an archive of the highest quality entities so far is kept up to date. After assessing the fitness of all the entities in a particular generation, the fitness of all archive entities of the previous generation and of all entities of this generation are compared, and the archive is adjusted accordingly.

Subsequently, entities are selected from this archive using a predefined *selection* operator. Unless specified otherwise, we will use the

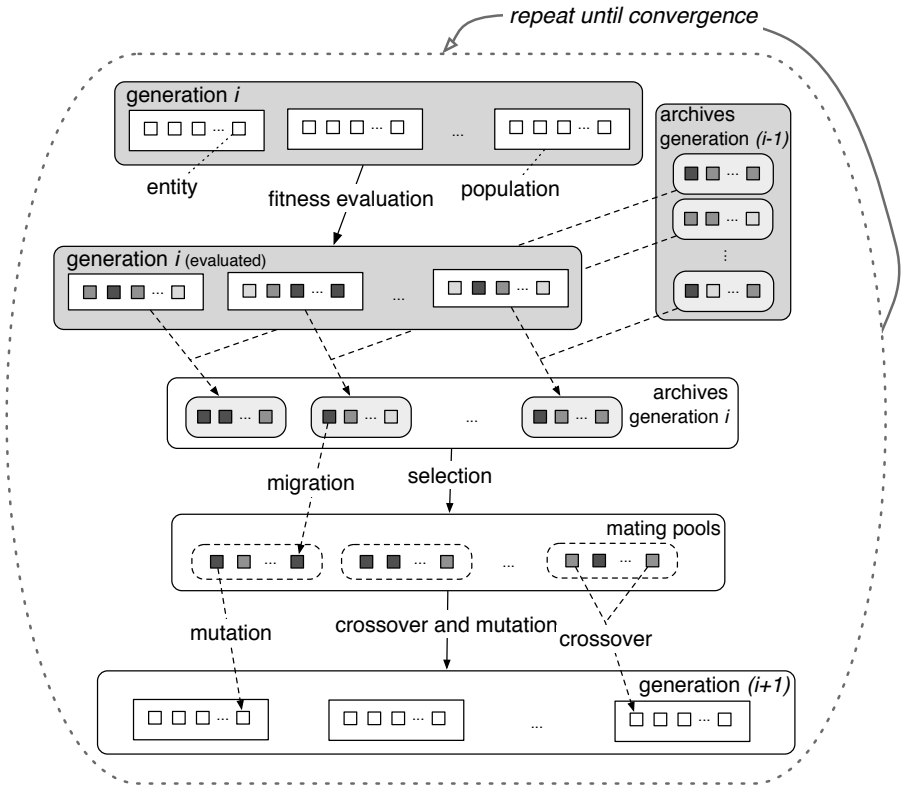


Figure A.5: The sequence of steps performed by a genetic algorithm iteratively until convergence is detected.

binary tournament selection technique which retains the entity with the highest fitness from a random pair of archive entities in each selection round. The resulting set of selected entities then forms the so-called *mating pool* of entities on which the crossover and mutation operators will be applied to generate the next generation of entities. When a generation consists of multiple populations an extra *migration* step is done, in which a number of archive entities are exchanged between the different populations. Although this also helps to avoid getting stuck in local optimal in particular populations, the overhead of evaluating a larger number of entities in multiple populations may outweigh its benefit in terms of entity quality.

This iterative process of evolving generation after generation of candidate solutions represented by entities ends when a state of *convergence* is detected, or when the maximum number of generations is

reached. There are various ways of defining convergence, or equivalently, the lack of progress in terms of quality of candidate solutions. One common way is to track the fitness of the best entity in each generation, and assume convergence when this fitness score shows little or no improvement across generations. Usually this involves a preset threshold value, which represents the absolute or relative difference between fitness scores. Often, the convergence criterion requires multiple subsequent generations without detecting (sufficient) progress in one or all of the populations.

Figure A.5 summarizes the various steps performed by a genetic algorithm, clarifies the definitions above and illustrates the use of the various operators.

A.2.5 Multi-objective evolutionary searching

In traditional genetic algorithms, a single objective is evaluated by the fitness function, and is the target to either maximize or minimize. In the case of multi-objective evolutionary algorithms, multiple objectives are considered at the same time, and the fitness function evaluates the quality of an entity along each of these objectives. Using a mechanism based on entity dominance, a single value representing the overall quality of the entity is computed. In this dissertation, we used the SPEA2 multi-objective evolutionary algorithm [113] whenever multiple objectives are of interest.

Appendix B

Benchmark Suites

B.1 BioMetricsWorkload

Table B.1: Overview of the workloads in the BiometricsWorkload benchmark suite. All workloads are run using the `s100` inputs.

workload	*input*	*# 100 M intervals*	*description*
hand		10,172	multi-layer perceptron (MLP) classification to identify handwriting
face	feret	9,899	statistical methods for face recognition
	scraps	4,236	
finger		24,365	neural network based fingerprint pattern classification
voice		2,834	voice recognition
gait	background	163	identification of people from gait
	difference	2,023	
	fast_similarity	1,120	

B.2 BioPerf

Table B.2: Overview of the workloads in the BioPerf benchmark suite. All workloads are run using the `medium` inputs.

workload	*input*	*# 100 M intervals*	*description*
Blast	nucleotide	1	searching of sequence databases
	protein	1,537	for local alignments
Fasta	fasta34	7,891	local similarity search in
	ssearch34	174,744	sequence databases
Clustalw		2,359	multiple sequence alignment
Hmmer	hmmpfam	4,201	aligning multiple sequences
	hmmsearch	1	using hidden Markov models
T-Coffee		2,286	multiple sequence alignment
Glimmer		377	finding genes in microbial DNA
Phylip	dnapenny	11,473	inferring phylogenies
	promlk	957	
Grappa		33,440	phylogeny reconstruction
Ce		36	finding structural similarities between pairs of proteins
Predator		5,329	finding protein structures

B.3 MediaBench II

Table B.3: Overview of the workloads in the MediaBenchII benchmark suite.

workload	*input*	*# 100 M intervals*	*description*
h263	decode	9	video coding
	encode	206	
h264	decode	33	motion-compensation-based
	encode	1,373	video coding
jpeg	decode	1	lossy image compression
	encode	1	
jpeg2000	decode	4	wavelet-based image compression
	encode	8	
mpeg2	decode	10	audio and video coding
	encode	205	
mpeg4	decode	2	audio and video coding
	encode	19	
mpeg4-mmx	decode	2	audio and video coding
	encode	5	(using MMX/SSE)

B.4 SPEC CPU2000

Table B.4: Overview of the integer workloads in the SPEC CPU2000 benchmark suite (SPECint2000).

workload	*input*	*# 100 M intervals*	*description*
bzip2	graphic	1,157	compression
	program	1,008	
	source	803	
crafty		1,421	game playing: chess
eon	cook	629	computer visualization
	kaijya	815	
	rushmeier	466	
gap		1,906	group theory, interpreter
gcc	166	267	C prog. language compiler
	200	725	
	expr	77	
	integrate	81	
	scilab	400	
gzip	graphic	699	compression
	log	301	
	program	1,276	
	random	556	
	source	633	
mcf		577	combinatorial optimization
parser		3,199	word processing
perlbmk	diffmail	312	Perl programming language
	makerand	11	
	perfect	198	
	splitmail-535	534	
	splitmail-704	565	
	splitmail-850	1,086	
	splitmail-957	937	
twolf		2,981	place and route simulator
vortex	lendian1	1,063	object-oriented database
	lendian2	1,195	
	lendian3	1,183	
vpr	place	928	FPGA circuit placement/routing
	route	706	

Table B.5: Overview of the floating-point workloads in the SPEC CPU2000 benchmark suite (SPECfp2000).

workload	*input*	*# 100 M intervals*	*description*
ammp		3,210	computational chemistry
applu		3,954	partial differential equations
apsi		4,263	meteorology: pollutant distribution
art	110	539	image recognition / neural networks
	470	594	
equake		1,206	seismic wave propagation simulation
facerec		2,883	image processing: face recognition
fma3d		2,754	finite-element crash simulation
galgel		3,013	computational fluid dynamics
lucas		2,608	number theory / primality testing
mesa		2,398	3-D graphics library
mgrid		4,627	multi-grid solver: 3D potential field
sixtrack		5,676	nuclear physics accelerator design
swim		2,218	shallow water modeling
wupwise		3,645	physics / quantum chromodynamics

B.5 SPEC CPU2006

Table B.6: Overview of the integer workloads in the SPEC CPU2006 benchmark suite (SPECint2006).

workload	*input*	*# 100 M intervals*	*description*
astar	BigLakes	4,202	path-finding algorithms
	river	8,495	
bzip2	chicken	1,787	compression
	combined	3,404	
	liberty	3,089	
	program	5,437	
	source	4,258	
	text	5,977	
gcc	166	808	C compiler
	200	1,521	
	c-typechk	1,458	
	cp-decl	1,093	
	expr	1,194	
	expr2	1,623	
	g23	1,894	
	s04	1,799	
	scilab	584	
gobmk	13x13	2,290	artificial intelligence: Go
	nngs	6,089	
	score2	3,156	
	trevorc	2,284	
	trevord	3,265	
h264ref	foreman_baseline	5,054	video compression
	foreman_main	2,907	
	sss	26,735	
hmmer	nph3	8,953	search gene sequence
	retro	20,167	
libquantum		24,139	physics / quantum computing
mcf		3,891	combinatorial optimization
omnetpp		6,004	discrete event simulation
perlbench	checkspam	10,955	Perl programming language
	diffmail	3,844	
	splitmail	6,911	
sjeng		24,086	artificial intelligence: chess
xalancbmk		11,257	XML processing

Table B.7: Overview of the floating-point workloads in the SPEC CPU2006 benchmark suite (SPECfp2006).

workload	*input*	*# 100 M intervals*	*description*
bwaves		37,361	fluid dynamics
cactusADM		27,800	physics / general relativity
calculix		75,380	structural mechanics
dealII		19,114	finite element analysis
gamess	cytosine h2ocu2+ triazolium	10,854 7,880 34,313	quantum chemistry
GemsFDTD		26,262	computational electromagnetics
gromacs		20,017	biochemistry / molecular dynamics
lbm		12,770	fluid dynamics
leslie3d		42,711	fluid dynamics
milc		11,733	physics / quantum chromodynamics
namd		25,891	biology / molecular dynamics
povray		10,014	image ray-tracing
soplex	pds-50 ref	3,746 3,876	linear programming, optimization
sphinx3		32,398	speech recognition
tonto		30,595	quantum chemistry
wrf		42,689	weather
zeusmp		20,386	physics / CFD

Appendix C

Tools

C.1 Hardware

C.1.1 Intel Xeon L5420 systems (Core)

For several experiments in this thesis we relied on the cluster of computer systems nicknamed `gengar`, which is part of the Ghent University supercomputer. The cluster consists of 194 systems each containing two dual socket quad-core Intel Xeon L5420 processors (8 cores in total). The most important details are shown in Table C.1.

Table C.2 lists the hardware performance counter events used to measure the most commonly used performance metrics in workload characterization studies (see Chapter 2). These events were measured in user-space using the `perfex` tool that comes with the `perfctr` Linux kernel patch [1].

C.1.2 Intel Xeon L5520 systems (Nehalem)

Another cluster which is part of the Ghent University supercomputer, nicknamed `gastly`, was used for selected experiments. This cluster consists of 80 systems each containing two dual socket quad-core Intel Xeon L5520 processors (8 cores in total). The most important details are listed in Table C.3.

[1] `http://user.it.uu.se/~mikpe/linux/perfctr/`

Table C.1: Hardware details for the Intel Xeon L5420 systems.

	parameter	*value*
processor		
	type	Intel Xeon L5420 (x2)
	details	dual socket quad-core
	architecture	Intel Core
	technology	45-nm
	clock frequency	2.5 GHz
	L1 instruction cache	32 kB
	L1 data cache	32 kB
	L2 cache (per dual-core)	6 MB
memory		
	amount of physical mem.	16 GB (DDR2 PC-5300)
	amount of virtual mem.	32 GB
local disk		
	size	146 GB (SAS)
	speed	10,000 RPM
OS		
	version	Scientific Linux 5.3
	Linux kernel version	2.6.18
	notes	kernel patched with perfctr 2.6.38

C.2 Software

This section provides information on two system software tools we experimented with in this dissertation.

C.2.1 GCC

The GNU Compiler Collection (GCC) is an open source software package, which includes static compilers for various programming languages including C, C++, Objective-C, Fortran, Java and Ada, as well as runtime libraries for these languages. It supports a wide variety of hardware platforms, including x86, both 32-bit and 64-bit, ARM, Alpha, PowerPC, MIPS and SPARC. GCC is the default compiler in most GNU Linux operation system distributions, and is therefore widely used in industry and academia. More information about GCC is available at `http://gcc.gnu.org`.

Table C.2: Hardware performance counter events used on Intel Xeon L5420 systems, with event numbers as specified in the Intel Software Developer's Manual, Volume 3B (Appendix A.1).

nr	*description*	*event*
(1)	number of dynamic instructions retired	`INST_RETIRED.ANY_P`
(2)	number of core cycles executed	`CPU_CLK_UNHALTED.CORE_P`
(3)	average number of cycles per instr.	derived from (1) and (2)
(4)	average number of L1 data cache misses per instr.	`MEM_LOAD_RETIRED.L1D_MISS`
(5)	average number of L1 instruction cache cache misses per instr.	`L1I_MISSES`
(6)	average number of L2 cache misses per instr.	`MEM_LOAD_RETIRED.L2_MISS`
(7)	average number of miss-predicted branches per instr.	`BR_INST_RETIRED.MISPRED`
(8)	average number of data TLB misses per instr.	`MEM_LOAD_RETIRED.DTLB_MISS`
(9)	average number of instruction TLB misses per instr.	`ITLB_MISS_RETIRED`

C.2.2 Jikes RVM

Jikes RVM (Research Virtual Machine) is an open source Java Virtual Machine targeted towards providing a vehicle for research on the frontiers of virtual machine technologies, such as garbage collection, dynamic compilation, adaptive optimization, thread scheduling and synchronization. More information about Jikes RVM is available at `http://www.jikesrvm.org`.

Table C.3: Hardware details for the Intel Xeon L5520 systems.

	parameter	*value*
processor		
	type	Intel Xeon L5520 (x2)
	details	dual socket quad-core
	architecture	Intel Nehalem
	technology	45-nm
	clock frequency	2.27 GHz
	L1 instruction cache	32 kB
	L1 data cache	32 kB
	L2 cache (per core)	256 kB
	L3 cache (per dual-core)	8 MB
memory		
	amount of physical mem.	12 GB
	amount of virtual mem.	12 GB
local disk		
	size	110 GB
	speed	10,000 RPM
OS		
	version	Scientific Linux 5.4
	Linux kernel version	2.6.18
	notes	kernel patched with perfctr 2.6.39

Bibliography

[1] F. Agakov, E. Bonilla, J. Cavazos, B.Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 295–305, March 2006.

[2] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 7–18, February 2003.

[3] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Compilation order matters: Exploring the structure of the space of compilation sequences using randomized search algorithms. In *Proceedings of the ACM SIGPLAN Symposium on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 231–239, June 2004.

[4] E. Alpaydin. *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2004.

[5] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.

[6] M. Annavaram, R. Rakvic, M. Polito, J. Bouguet, R. Hankins, and B. Davies. The fuzzy correlation between code and performance predictability. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, pages 93–104, December 2004.

[7] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 47–65, October 2000.

[8] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machine. *Proceedings of the IEEE*, 93(2):449–466, February 2005.

[9] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 111–129, October 2002.

[10] R. Azimi, M. Stumm, and R. W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th annual international conference on Supercomputing (ISC)*, pages 101–110, New York, NY, USA, 2005. ACM.

[11] T. Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, Oxford, UK, 1996.

[12] D. A. Bader, Y. Li, T. Li, and V. Sachdeva. BioPerf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization (IISWC)*, pages 163–173, October 2005.

[13] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33th Annual International Symposium on Microarchitecture (MICRO)*, pages 245–257, December 2000.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT-2008)*, October 2008.

[15] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z.

Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, October 2006.

[16] S.M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. . Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA-2006)*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[17] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback-Directed Compilation, in Conjunction with the Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, October 1998.

[18] M. Brehob and R. Enbody. An analytical model of locality and caching. Technical Report MSU-CSE-99-31, Michigan State University, August 1999.

[19] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 185–197, March 2007.

[20] J. Cavazos and M. O'Boyle. Automatic tuning of inlining heuristics. In *Proceedings of the Supercomputing Conference on High Performance Networking and Computing*, page 14, November 2005.

[21] J. Cavazos and M. O'Boyle. Method-specific dynamic compilation using logistic regression. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 229–240, October 2006.

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

[23] I. K. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 128–137, October 1996.

[24] D. Chiou, D. Sunwoo, J. Kim, N. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. FPGA-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-2007)*, pages 249–261, Washington, DC, USA, 2007. IEEE Computer Society.

[25] C.-B. Cho, A. V. Chande, Y. Li, and T. Li. Workload characterization of biometric applications on Pentium 4 microarchitecture. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization (IISWC)*, pages 76–86, October 2005.

[26] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *Proceedings of the Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*, November 1999.

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

[28] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, May 1999.

[29] Standard Performance Evaluation Corporation. SPECjvm98 benchmarks. `http://www.spec.org/jvm98`.

[30] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. Wiley, 2001.

[31] K. Deb and S. Agrawal. Understanding interactions among genetic algorithm parameters. In *Foundations of Genetic Algorithms 5*, pages 265–286. Morgan Kaufmann, 1998.

[32] A. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 233–244, May 2002.

[33] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 220–231, October 2003.

[34] L. Eeckhout, A. Georges, and K. De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 169–186, October 2003.

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

[36] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5, February 2003. `http://www.jilp.org/vol5`.

[37] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5, February 2003. `http://www.jilp.org/vol5`.

[38] M. Ekman and P. Stenström. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 89–99, March 2005.

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

[40] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 22nd Annual International Symposium on Microarchitecture (MICRO)*, pages 236–245, December 1992.

[41] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf. Mediabench II video: Expediting the next generation of video systems research. *Microprocess. Microsyst.*, 33(4):301–318, 2009.

[42] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. Quick and practical run-time evaluation of multiple program optimizations. *Transactions on High Performance Embedded Architectures and Compilation Techniques (HiPEAC)*, 1(1):13–31, 2006.

[43] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 57–76, October 2007.

[44] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in Java workloads. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Languages, Applications and Systems (OOPSLA)*, pages 270–287, October 2004.

[45] S. V. Gheorghita, H. Corporaal, and T. Basten. Iterative compilation for energy reduction. *Journal of Embedded Computing*, 1(4):509–520, July 2005.

[46] C. Gondro and B.P. Kinghorn. A simple genetic algorithm for multiple sequence alignment. *Genetics and Molecular Research*, 6(4):964–982, 2007.

[47] E. Granston and A. Holler. Automatic recommendation of compiler options. In *Proceedings of the Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*, December 2001.

[48] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[49] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Optimizing general purpose compiler optimization. In *Proceedings of the International Conference on Computing Frontiers*, pages 180–188, May 2005.

[50] M. Haungs, P. Sallee, and M. Farrens. Branch transition rate: A new metric for improved branch classification analysis. In *Proceedings on the Internatioan Symposium on High-Performance Computer Architecture (HPCA)*, pages 241–250, 2000.

[51] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[52] K. Hoste and L. Eeckhout. Characterizing the unique and diverse behaviors in existing and emerging general-purpose and domain-specific benchmark suites. In *Proceedings of the Annual International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 157–168, April 2006.

[53] K. Hoste and L. Eeckhout. Comparing benchmarks using key microarchitecture-independent characteristics. In *Proceedings of the 2006 IEEE International Symposium on Workload Characterization (IISWC)*, pages 83–92, October 2006.

[54] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72, May 2007.

[55] K. Hoste and L. Eeckhout. COLE: Compiler optimization level exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 165–174, April 2008.

[56] K. Hoste and L. Eeckhout. A methodology for analyzing commercial processor performance numbers. *IEEE Computer*, 42(10):70–76, 2009.

[57] K. Hoste, A. Georges, and L. Eeckhout. Automated just-in-time compiler tuning. In *Proceedings of the 8th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 62–72, New York, NY, USA, 2010. ACM.

[58] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 2006 International Conference*

on Parallel Architectures and Compilation Techniques (PACT), pages 114–122, September 2006.

[59] M. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 157–168, June 2003.

[60] Intel. Moore's law. `http://www.intel.com/technology/mooreslaw`.

[61] Intel. MSC4. `http://www.intel.com/Assets/PDF/Manual/msc4.pdf`.

[62] K. Ishizaki, Takeuchi M., K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu, and T. Nakatani. Effectiveness of cross-platform optimizations for a Java just-in-time compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 187–204, October 2003.

[63] Z. Jin and A. C. Cheng. Implantbench: Characterizing and projecting representative benchmarks for emerging bioimplantable computing. *IEEE Micro*, 28(4):71–91, 2008.

[64] R. A. Johnson and D. W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, fifth edition, 2002.

[65] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13(4):455–492, December 1998.

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

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

[68] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN Conference on Pro-*

gramming Language Design and Implementation (PLDI), pages 171–182, June 2004.

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

[70] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *Proceedings of the 2004 International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 57–67, March 2004.

[71] Y. Li and T. Li. BioInfoMark: A bioinformatic benchmark suite for computer architecture research. Technical report, ECE, University of Florida, January 2005.

[72] Y. Li, K. C. Ng, D.J. Murray-Smith, G.J. Gray, and K. C. Sharman. Genetic algorithm automated approach to design of sliding mode control systems. *International Journal of Control*, 63:721–739, 1996.

[73] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[74] Y. Lin and J. Zhang. An isoline genetic algorithm. In *Proceedings of the Eleventh conference on Congress on Evolutionary Computation (CEC)*, pages 2002–2007, 2009.

[75] S. Liu and J. Gaudiot. Potential impact of value prediction on communication in many-core architectures. *IEEE Transactions on Computers*, 58:759–769, 2009.

[76] D. Maier, P. Ramarao, M. Stoodley, and V. Sundaresan. Experiences with multithreading and dynamic class loading in a Java just-in-time compiler. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 87–97, March 2006.

[77] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of The IEEE International Symposium on Workload Characterization (IISWC-2008)*, pages 35–46, September 2008.

[78] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[79] P. Nagpurkar, C. Krintz, and T. Sherwood. Phase-aware remote profiling. In *Proceedings of the International Conference on Code Generation and Optimization (CGO)*, pages 191–202, March 2005.

[80] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *Proceedings of the 2006 IEEE International Symposium on Workload Characterization (IISWC)*, pages 182–188, 2006.

[81] D. Novillo. GCC - yesterday, today and tomorrow. `http://www.airs.com/dnovillo/Papers/hipeac2007.pdf`, January 2007.

[82] M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *Java Virtual Machine Research and Technology Symposium (JVM)*, pages 1–12, April 2001.

[83] Z. Pan and R. Eigenmann. Fast, automatic, procedure-level performance tuning. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 173–181, September 2006.

[84] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO)*, pages 81–93, December 2004.

[85] A. Phansalkar and L. K. John. Performance prediction using program similarity. In *Proceedings of the 2006 SPEC Benchmark Workshop*, January 2006.

[86] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring program similarity: Experiments with SPEC CPU benchmark suites. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 10–20, March 2005.

[87] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA'07)*, pages 412–423, 2007.

[88] A.J Poovey, T.M Conte, M Levy, and S Gal-On. A benchmark characterization of the EEMBC benchmark suite. *IEEE Micro*, 29(5):18–29, 2009.

[89] R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, November 1996.

[90] V. Sachdeva, E. Speight, M. Stephenson, and L. Chen. Characterizing and improving the performance of bioinformatics workloads on the POWER5 architecture. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 89–97, 2007.

[91] D. Shelepov, J.C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. HASS: a scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev.*, 43(2):66–75, 2009.

[92] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 165–176, October 2004.

[93] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, October 2002.

[94] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93, 2003.

[95] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 336–347, June 2003.

[96] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, July 2002.

[97] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations

for a Java just-in-time compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):732–785, July 2005.

[98] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 204–215, 2003.

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

[100] H. Vandierendonck and K. De Bosschere. Experiments with subsetting benchmark suites. In *Proceedings of the Seventh Annual IEEE International Workshop on Workload Characterization (WWC)*, pages 55–62, October 2004.

[101] K. Weicker and N. Weicker. Basic principles for understanding evolutionary algorithms. *Fundam. Inf.*, 55(3-4):387–403, 2002.

[102] R. P. Weicker. An overview of common benchmarks. *IEEE Computer*, 23(12):65–75, December 1990.

[103] M. Wolfe. Compilers and more: Gloptimizations. HPCwire: see `http://www.hpcwire.com/features/ Compilers_and_More_Gloptimizations.html`, November 2007.

[104] H. Wu, E. Park, M. Kaplarevic, and Y. Zhang. Dynamic optimization option search in GCC. In *Proceedings of the GCC Developers Summit 2007*, June 2007.

[105] T. Y. Yeh, P. Faloutsos, S. J. Patel, and G. Reinman. Parallax: an architecture for real-time physics. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA)*, pages 232–243, 2007.

[106] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 124–134, May 1992.

[107] J. J. Yi, D. J. Lilja, and D. M. Hawkins. A statistically rigorous approach for improving simulation methodology. In *Proceedings*

of the Ninth International Symposium on High Performance Computer Architecture (HPCA), pages 281–291, February 2003.

[108] J. J. Yi, R. Sendag, L. Eeckhout, A. Joshi, D. J. Lilja, and L. K. John. Evaluating benchmark subsetting approaches. In *Proceedings of the 2006 IEEE International Symposium on Workload Characterization (IISWC)*, pages 93–104, October 2006.

[109] J. J. Yi, H. Vandierendonck, L. Eeckhout, and D. J. Lilja. The exigency of benchmark and compiler drift: Designing tomorrow's processors with yesterday's tools. In *Proceedings of the 20th ACM International Conference on Supercomputing (ICS)*, pages 75–86, June 2006.

[110] M. Zhao, B. R. Childers, and M. L. Soffa. Predicting the impact of optimizations for embedded systems. In *Proceedings of the ACM SIGPLAN Symposium on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–11, June 2003.

[111] Y. Zhong, S. G. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *Proceedings of the Twelfth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2003.

[112] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 129–142, New York, NY, USA, 2010. ACM.

[113] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm. Technical Report TIK-Report 103, Swiss Federal Institute of Technology (ETH) Zurich, May 2001.

[114] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength perato approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, November 1999.