Werklastgeneratie voor de prestatie-evaluatie van microprocessors

Workload Generation for Microprocessor Performance Evaluation

Luk Van Ertvelde

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 2010 - 2011

UNIVERSITEIT
GENT

ISBN 978-90-8578-395-4
NUR 980, 958
Wettelijk depot: D/2010/10.500/71

To my friends, family, and teachers.

Dankwoord

De weg naar een doctoraat is niet altijd vanzelfsprekend. Het is onmogelijk om de eindbestemming te bereiken zonder de actieve hulp en steun van meerdere personen. Graag wil ik al degene die mij de afgelopen jaren hebben bijgestaan van harte bedanken.

Als eerste wil ik in het bijzonder mijn promotor prof. Lieven Eeckhout bedanken voor de succesvolle samenwerking. Hij gaf de juiste richting aan wanneer ik de weg kwijt was. Zijn actieve ondersteuning bij het schrijven van artikels bleek van onschatbare waarde. Het is dankzij zijn ervaring en expertise dat ik onderzoeksresultaten heb mogen voorstellen op belangrijke internationale conferenties. Zeven dagen op zeven, vierentwintig uur per dag, bood hij antwoord op al mijn vragen. Lieven, dankjewel!

Ik zou verder ook prof. Koen De Bosschere willen bedanken. Zijn fascinerende lessen hebben mijn interesse in de wereld van de computerarchitectuur opgewekt. Hij heeft mij aangeleerd hoe onderzoeksresultaten aantrekkelijk te presenteren.

Ik wens de andere leden van mijn examencommissie eveneens uitdrukkelijk te bedanken. Mijn dank gaat uit naar prof. Jan Van Campenhout, de voorzitter van ELIS, de vakgroep waar dit onderzoek is gebeurd. Zijn kritische opmerkingen, zowel tijdens de reviewvergaderingen als bij het nalezen van dit proefschrift, hebben bijgedragen tot de kwaliteit van mijn onderzoek. Ik dank ook prof. Serge Demeyer (Universiteit Antwerpen) en prof. Bart Dhoedt (Universiteit Gent) voor hun inspanning om dit werk te lezen en te beoordelen.

I would also like to thank the foreign members of my PhD committee for their effort to evaluate my thesis. Thanks to dr. Harish Patil (Intel, USA) and dr. Emre Ozer (ARM, United Kingdom) for their time to read my dissertation and come over to Ghent to serve on my committee. Your comments were invaluable. Your interest and appreciation gave me great satisfaction.

Ik bedank ook de andere professoren binnen ELIS voor hun feedback en suggesties na het geven van een interne presentatie. In het bijzonder dank ik prof. Dirk Stroobandt voor het organiseren van de KIS-verdedigingen, en prof. Erik D'Hollander voor het aanreiken van wijzers naar wetenschappelijke literatuur.

Ik wil ook mijn bureaugenoten Frederik, Juan, Stijn, Kenneth, Filip, Kenzo, Trevor en Jeroen bedanken voor de fijne werksfeer. Trevor wens ik in het bijzonder te bedanken voor het nalezen van mijn proefschrift, Kenzo voor de prettige discussies op vrijdagnamiddag. Ook mijn andere collega's van *-3* wens ik te bedanken voor de aangename momenten samen.

Het sociale aspect mag tijdens een doctoraat niet uit het oog worden verloren. Ik heb een aantal onvergetelijke donderdagavonden mogen beleven met Sean, Tim, Tom, Filip en Pieter. Dankjewel hiervoor! Dankjewel ook aan Kristof, Pieter, Kathleen en alle anderen voor de wekelijkse sportieve inspanningen samen.

Ik wens ook van de gelegenheid gebruik te maken om de mensen te bedanken die mij de (bijna) voorbije dertig jaar hebben gevormd. Ik denk dan in het bijzonder aan mijn ouders en broers, maar ook aan alle docenten die mij vol overgave hun kennis en inzicht hebben overgebracht.

Last but not least wens ik mijn vriendin Dagmar te bedanken. Dagmar kreeg het de laatste maanden voor het afwerken van deze thesis soms onnodig hard te verduren. Mijn humeur bleek meer dan eens afhankelijk van onderzoeksresultaten. Tijdens het schrijven van dit boek kon ik er niet altijd voor haar zijn, maar steeds kon ik op haar begrip en steun rekenen.

Luk Van Ertvelde
Gent, 6 december 2010

Examencommissie

Prof. Daniël De Zutter, voorzitter
Prodecaan, 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. Serge Demeyer
Vakgroep LORE, Departement Wiskunde & Informatica
Universiteit Antwerpen

Prof. Bart Dhoedt
Vakgroep INTEC, Faculteit Ingenieurswetenschappen
Universiteit Gent

Dr. Emre Özer
ARM, Cambridge
United Kingdom

Dr. Harish G. Patil
Intel, Hudson, MA
USA

Leescommissie

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

Prof. Serge Demeyer
Vakgroep LORE, Departement Wiskunde & Informatica
Universiteit Antwerpen

Prof. Bart Dhoedt
Vakgroep INTEC, Faculteit Ingenieurswetenschappen
Universiteit Gent

Dr. Emre Özer
ARM, Cambridge
United Kingdom

Dr. Harish G. Patil
Intel, Hudson, MA
USA

Samenvatting

De vooruitgang van de microprocessor is de laatste decennia exponentieel verlopen: van eenvoudige processors die instructies in programmavolgorde uitvoeren tot complexe processors die per klokslag meerdere instructies in een mogelijk andere volgorde kunnen uitvoeren. Door deze steeds hogere complexiteit zijn programma's om de prestatie van (nieuwe) processors te evalueren onmisbaar geworden, en bijgevolg hebben meerdere organisaties gestandaardiseerde verzamelingen van evaluatieprogramma's[1] ontwikkeld en vrijgegeven. Hoewel dit het prestatie-evaluatieproces heeft gestandardiseerd, staan computerarchitecten en computeringenieurs nog steeds voor een aantal belangrijke uitdagingen. Bij het selecteren van een goed evaluatieprogramma kunnen we drie overkoepelende uitdagingen/problemen identificeren die uiteindelijk aanleiding hebben gegeven tot dit onderzoekswerk.

1. De evaluatieprogramma's moeten in de eerste plaats representatief zijn voor de programma's die het (toekomstige) systeem moet uitvoeren, maar het is niet altijd mogelijk om zo'n representatieve verzameling van evaluatieprogramma's te selecteren. We kunnen hiervoor drie redenen onderscheiden. Een eerste reden is dat gestandaardiseerde verzamelingen van evaluatieprogramma's typisch worden ontwikkeld op basis van openbronprogramma's omdat software-ontwikkelaars terughoudend zijn om hun eigendomsprogramma's te verspreiden. Het probleem hierbij is dat openbronprogramma's een ander prestatiegedrag kunnen vertonen dan commerciële applicaties, of een bepaalde doelapplicatie. Bijgevolg kan het gebruik van openbronprogramma's leiden tot

[1] We gebruiken 'evaluatieprogramma' als Nederlandse vertaling van het Engelse woord '*benchmark*', en 'verzameling van evaluatieprogramma's' als vertaling van '*benchmark suite*'.

verkeerde aankoopbeslissingen of tot een suboptimaal ontwerp bij het verkennen van de ontwerpruimte van een processor. Een tweede oorzaak is dat beschikbare evaluatieprogramma's vaak verouderd zijn omdat de applicatieruimte voortdurend evolueert en het ontwikkelen en onderhouden van evaluatieprogramma's (op basis van die applicaties) veel tijd in beslag neemt. Een laatste reden is dat men evaluatieprogramma's niet zomaar op basis van bestaande applicaties kan ontwikkelen in het geval men toekomstige werklasten wenst te modelleren.

2. Een tweede uitdaging die aanleiding heeft gegeven tot dit onderzoekswerk is het terugdringen van het aantal instructies dat een evaluatieprogramma uitvoert. Hedendaagse evaluatieprogramma's voeren honderden miljarden instructies uit om toekomstige processorontwerpen zinvol te belasten, maar dit belemmert het gebruik van gedetailleerde (cyclusgetrouwe) simulatoren om de (micro)architecturale ontwerpruimte te verkennen. Het simuleren van één zo'n programma neemt immers meerdere dagen tot zelfs maanden in beslag, en bovendien is dit voor de evaluatie van slechts één punt in de (micro)architecturale ruimte. Dit kan uiteindelijk leiden tot vertragingen bij het op de markt brengen van nieuwe processors.

3. De laatste uitdaging is het ontwikkelen van evaluatieprogramma's die kunnen worden aangewend voor zowel de exploratie van verschillende (micro)architecturen als de exploratie van verschillende compilers en compileroptimalisaties. Bestaande evaluatieprogramma's voldoen aan deze voorwaarde maar dit is typisch niet het geval voor evaluatieprogramma's die automatisch worden gegenereerd of gereduceerd. De reden hiervoor is dat bestaande technieken die automatisch evaluatieprogramma's genereren of reduceren opereren op assemblerniveau, waardoor de resulterende programma's niet aangewend kunnen worden voor de evaluatie van bijvoorbeeld instructiesetuitbreidingen of nieuwe compilertechnieken.

Samengevat willen we dus beschikken over korte maar representatieve evaluatieprogramma's die bovendien gebruikt kunnen worden voor zowel de evaluatie van verschillende (micro)architecturen als voor de evaluatie van compilers en hun optimalisaties. In dit onderzoekswerk stellen we een aantal technieken voor die het mogelijk

moeten maken om dergelijke evaluatieprogramma's te genereren.

Codemutatie. We stellen eerst codemutatie voor om de verspreiding van eigendomssoftware over verschillende partijen (bedrijven en universiteiten) mogelijk te maken met als eindoel het beschikken over representatievere evaluatieprogramma's. Codemutatie is een nieuwe methode om evaluatieprogramma's te genereren op basis van eigendomssoftware. De gegenereerde evaluatieprogramma's (of evaluatiemutanten) verbergen de oorspronkelijke programma-algoritmen maar vrijwaren de prestatie van het oorspronkelijke programma. We buiten hiervoor twee observaties uit: (i) cachemissers en foutief voorspelde paden hebben een bepalende impact op de prestatie van hedendaagse microprocessors en (ii) meerdere variabelen vertonen een constant verloop tijdens de uitvoering van een programma. We doen dit meer concreet door oorspronkelijke programma-instructies te muteren die geen invloed uitoefenen op het controleverloop en/of dataverloop. We berekenen hiervoor eerst programmasneden van geheugenoperaties en/of controleverloopoperaties. Vervolgens trimmen we deze sneden door gebruik te maken van profielinformatie over het verloop van variabelen en sprongen. Uiteindelijk overschrijven (muteren) we de instructies die geen deel uitmaken van deze gereduceerde sneden door instructies met gelijkaardige prestatiekarakteristieken. Het resulterende programma fungeert dan als alternatief voor het eigendomsprogramma, en kan worden verspreid (zonder vrijgave van de originele programma-algoritmen) met het oog op prestatie-evaluatie door derden.

De proefondervindelijke resultaten (opgemeten voor een verzameling van SPEC CPU2000 en MiBench evaluatieprogramma's) bevestigen dat codemutatie een effectieve aanpak is die tot 90 procent van de statische instructies muteert, tot 50 procent van de dynamische instructies, en tot 35 procent van de dynamische data-afhankelijkheden. Bovendien zijn de prestatieresultaten van de evaluatiemutanten sterk gelijkaardig aan de prestatieresultaten van de oorspronkelijke programma's.

Bedrijven die (ingebedde) microprocessors ontwikkelen en bedrijven die diensten via het internet aanbieden kunnen het meeste voordeel halen uit onze codemutatietechniek. Dergelijke bedrijven zijn immers terughoudend om hun eigendomssoftware te distribueren. Wij stellen codemutatie voor om hun eigendomssoftware te muteren

en vervolgens de gemuteerde software aan te wenden om prestatie-evaluatie door derden mogelijk te maken. Codemutatie heeft ook het potentieel om de samenwerking tussen industrie en universiteiten te versterken; dit kan leiden tot representatievere prestatiemetingen in universiteiten, en uiteindelijk zinvollere onderzoeksrichtingen.

Bemonsterde simulatie: NSL-BLRL. Codemutatie op zich biedt geen antwoord op de extreem lange simulatietijd van hedendaagse (evaluatie)programma's. We kunnen hiervoor wel beroep doen op bemonsterde simulatie. Het idee van bemonsterde simulatie is om slechts een deel (het zogenaamde monster) van een volledige programma-uitvoering gedetailleerd te simuleren en zo snelheidswinst te behalen. Merk op dat een monster hierbij kan bestaan uit één of meerdere monstereenheden. Een belangrijk probleem bij bemonsterde simulatie is de onbekende microarchitecturale toestand (toestand van de caches, sprongvoorspeller, ...) aan het begin van elke monstereenheid; gerefereerd in de literatuur als het koudestartprobleem.

Om het koudestartprobleem aan te pakken, stellen we een nieuwe opwarmmethode voor: NSL-BLRL. Deze methode bouwt verder op *No-State-Loss* (NSL) en *Boundary Line Reuse Latency* (BLRL) om de kost van cyclusgetrouwe cachesimulatie in bemonsterde simulatie te minimaliseren. Het achterliggende idee is om de toestand van de cache aan het begin van een monstereenheid efficiënt te initialiseren door het inladen van een ingekorte NSL-stroom. NSL overloopt de voormonstereenheid[2] en houdt de laatste referentie naar elke unieke geheugenlocatie bij. Deze NSL-stroom wordt dan ingekort door gebruik te maken van BLRL-informatie: we bepalen tot waar in de voormonstereenheid we moeten teruggaan om de cache (voldoende) nauwkeurig op te warmen.

Deze hybride aanpak heeft een aantal belangrijke voordelen t.o.v. voorgaand onderzoek. We bereiken aanzienlijke simulatieversnellingen t.o.v. BLRL en belangrijke data-opslagreducties t.o.v. NSL. Bovendien is NSL-BLRL breder toepasbaar dan de recent voorgestelde MHS-aanpak of de TurboSMARTS-aanpak omdat NSL-BLRL onafhankelijk is van de cacheblokgrootte.

[2]De voormonstereenheid (Eng.: *pre-sampling unit*) bestaat uit de instructies tussen twee monstereenheden (of tussen het begin van het programma en de eerste monstereenheid).

Benchmarksynthese. Codemutatie kan in combinatie met bemonsterde simulatie gebruikt worden om korte evaluatieprogramma's te genereren die de originele programma-algoritmen verbergen. Hoewel dit een effectieve aanpak is, zijn er een aantal nadelen aan verbonden. Het is bijvoorbeeld niet mogelijk om deze evaluatieprogramma's te gebruiken voor de exploratie van verschillende instructiesetarchitecturen of voor de exploratie van verschillende compilers en compileroptimalisaties. Bovendien moet bij bemonsterde simulatie de simulator typisch aangepast worden om de (micro)architecturale toestand aan het begin van een monstereenheid te initialiseren. In het laatste luik van dit onderzoekswerk stellen we een raamwerk voor dat korte evaluatieprogramma's genereereert in een hogere programmeertaal, om zo te trachten antwoord te bieden op deze en de eerder geïdentificeerde problemen.

Het geïntroduceerde raamwerk genereert korte, onherkenbare maar toch representatieve evaluatieprogramma's, en omdat de programma's in een hogere programmeertaal worden gegenereerd, kunnen ze worden aangewend voor de exploratie van verschillende architecturen, compilers en compileroptimalisaties. De eerste component van het raamwerk profileert de (eigendoms)software (gecompileerd met een lage optimalisatievlag) om verschillende uitvoeringskarakteristieken op te meten. We maken hierbij gebruik van een nieuwe structuur die het controleverloop van een programma statistisch beschrijft. De tweede component van het raamwerk genereert op basis van het opgemeten profiel een synthetisch evaluatieprogramma in een hogere programmeertaal (C in dit geval). Voorgaand onderzoek beperkte zich tot de generatie van evaluatieprogramma's in assemblertaal. De gegenereerde synthetische evaluatieprogramma's hebben een aantal belangrijke eigenschappen. Allereerst zijn de evaluatieprogramma's representatief voor de eigendomssoftware op verschillende platformen (met verschillende instructiesetarchitecturen, microarchitecturen, en compilers en optimalisatieniveau's). Ook zijn de gegenereerde programma's korter dan de oorspronkelijke programma's (gemiddeld dertig keer korter, opgemeten voor een verzameling van MiBench evaluatieprogramma's). De laatste eigenschap is dat de synthetische programma's de algoritmen van de eigendomssoftware verbergen. We hebben dit geverifieerd door gebruik te maken van plagiaatdetectiesoftware.

Het raamwerk kan gebruikt worden bij verschillende toepassingen: het verspreiden van representatieve klonen van eigendomssoftware voor prestatie-evaluatiedoeleinden, het versnellen van gedetailleerde

simulaties, het modelleren van toekomstige werklasten, het consolideren van meerdere programma's in één synthetisch evaluatieprogramma, enz.

Conclusie. We hebben een aantal technieken voorgesteld om werklasten te genereren en/of reduceren. Codemutatie transformeert een eigendomsprogramma in een mutant die de intellectuele eigendom verbergt, en de prestatie vrijwaart. Bemonsterde simulatie kan gebruikt worden om gedetailleerde simulaties te versnellen. Benchmarksynthese verbergt de intellectuele eigendom van een programma en reduceert tevens de uitvoeringstijd.

Summary

Microprocessors have advanced exponentially over the years: from scalar in-order execution processors to complex superscalar out-of-order and multi-core processors. This ever-increasing microarchitectural complexity necessitates benchmark programs to evaluate the performance of a (new) microprocessor, hence, organizations such as SPEC, EEMBC, etc., released standardized benchmark suites. Although this has streamlined the process of performance evaluation, computer architects and engineers still face several important benchmarking challenges. We identify three overarching challenges in benchmark design, which gave rise to this research work.

1. Benchmarks should be representative of the (future) applications that are expected to run on the target computer system; however, it is not always possible to select a representative benchmark suite for at least three reasons. For one, standardized benchmark suites are typically derived from open-source programs because industry hesitates to share proprietary applications, and open-source programs have the advantage that they are portable across different platforms. The limitation though is that these benchmarks may not be representative of the real-world applications of interest. Second, existing benchmark suites are often outdated because the application space is constantly evolving and developing new benchmark suites is extremely time-consuming (and costly). Finally, benchmarks are modeled after existing applications that may be less relevant by the time the product hits the market.

2. Coming up with a benchmark that is short-running yet representative is another major challenge. Contemporary application benchmark suites like SPEC CPU2006 execute trillions of instructions in order to stress contemporary and future processors in a

meaningful way. If we also take into account that during microarchitectural research a multitude of design alternatives need to be evaluated, we easily end up with months or even years of simulation time. This may stretch the time-to-market of newly designed microprocessors. Hence, it is infeasible to simulate entire application benchmarks using detailed cycle-accurate simulators.

3. Finally, a benchmark should enable both (micro)architecture and compiler research and development. Although existing benchmarks satisfy this requirement, this is typically not the case for workload generation techniques that reduce the dynamic instruction count in order to address the simulation challenge. These techniques often operate on binaries which eliminates their utility for compiler exploration and instruction-set architecture exploration.

In summary, we would like to have a representative benchmark that is short-running and that can be used for both (micro)architecture and compiler research. In this research work, we propose three novel workload generation and reduction techniques that help to fulfill these challenges. In particular, code mutation addresses the proprietary nature of contemporary applications, while sampled simulation using NSL-BLRL reduces the long simulation times of contemporary benchmarks; finally, benchmark synthesis reduces simulation time and hides proprietary information in the reduced workloads.

Code mutation. We first propose code mutation to stimulate sharing of proprietary applications to third parties (academia and industry). Code mutation is a novel methodology that mutates a proprietary application to complicate reverse engineering so that it can be distributed as an application benchmark among several parties. These benchmark mutants hide the functional semantics of proprietary applications while exhibiting similar performance characteristics. We therefore exploit two observations: (i) miss events have a dominant impact on performance on contemporary microprocessors, and (ii) many variables of contemporary applications exhibit invariant behavior at run time. More specifically, we compute program slices for memory access operations and/or control flow operations trimmed through constant value and branch profiles. Subsequently, we mutate the instructions not appearing in these slices through binary rewriting. The end result

is a benchmark mutant that can serve as a proxy for the proprietary application during benchmarking experiments by third parties.

Our experimental results using SPEC CPU2000 and MiBench benchmarks show that code mutation is an effective approach that mutates (i) up to 90% of the binary, (ii) up to 50% of the dynamically executed instructions, and (iii) up to 35% of the at-run-time-exposed inter-operation data dependencies. In addition, the performance characteristics of the mutant are very similar to those of the proprietary application across a wide range of microarchitectures and hardware implementations.

Code mutation will mostly benefit companies that develop (embedded) microarchitectures and companies that offer (in-house built) services to remote customers. Such companies are reluctant to distribute their proprietary software. As an alternative, they can use mutated benchmarks as proxies for their proprietary software to help drive performance evaluation by third parties as well as guide purchasing decisions of hardware infrastructure. Being able to generate representative benchmark mutants without revealing proprietary information can also be an encouragement for industry to collaborate more closely with academia, i.e., it would make performance evaluation in academia more realistic and therefore more relevant for industry. Eventually, this may lead to more valuable research directions. In addition, developing benchmarks is both hard and time-consuming to do in academia, for which code mutation may be a solution.

Sampled simulation: NSL-BLRL. Code mutation conceals the intellectual property of an application, but it does not lend itself to the generation of short-running benchmarks. Sampled simulation on the other hand reduces the simulation time of an application significantly. The key idea of sampled simulation is to simulate only a small sample from a complete benchmark execution in a detailed manner (a sample consists of one or more sampling units). The performance bottleneck in sampled simulation is the establishment of the microarchitecture state (caches, branch predictor, etc) at the beginning of each sampling unit. The unknown microarchitecture starting image at the beginning of a sampling unit is often referred to as the cold-start problem.

We address the cold-start problem by proposing a new cache warmup method, namely NSL-BLRL which builds on No-State-Loss (NSL) and Boundary Line Reuse Latency (BLRL) for minimizing the

cost associated with cycle-accurate processor cache hierarchy simulation in sampled simulation. The idea of NSL-BLRL is to establish the cache state at the beginning of a sampling unit using a checkpoint that stores a truncated NSL stream. NSL scans the pre-sampling unit and records the last reference to each unique memory location. This is called the least-recently used (LRU) stream. This stream is then truncated to form the NSL-BLRL warmup checkpoint by inspecting the sampling unit for determining how far in the pre-sampling unit one needs to go back to accurately warm up the cache state for the given sampling unit.

This approach yields several benefits over prior work: substantial simulation speedups compared to BLRL (up to 1.4× under fast-forwarding and up to 14.9× under checkpointing) and significant reductions in disk space requirements compared to NSL (on average 30%), for a selection of SPEC CPU2000 benchmarks. Also, NSL-BLRL is more broadly applicable than the Memory Hierarchy State (MHS) and TurboSMARTS approaches because NSL-BLRL warmup is independent of the cache block size.

Benchmark synthesis. Although code mutation can be used in combination with sampled simulation to generate short-running workloads that can be distributed to third parties without revealing intellectual property, there are a number of limitations. The most important limitation is that this approach operates at the assembly level, and as a result, it cannot be used for compiler exploration and ISA exploration purposes. We therefore propose a novel benchmark synthesis framework that generates synthetic benchmarks in a high-level programming language.

The benchmark synthesis framework aims at generating small but representative benchmarks that can serve as proxies for other applications without revealing proprietary information; and because the benchmarks are generated in a high-level language, they can be used to explore the architecture and compiler space. The methodology to generate these benchmarks comprises two key steps: (i) profiling a real-world (proprietary) application (that is compiled at a low optimization level) to measure its execution characteristics, and (ii) modeling these characteristics into a synthetic benchmark clone. To capture a program's control flow behavior in a statistical way, we introduce a new structure: the Statistical Flow Graph with Loop information (SFGL).

We demonstrate good correspondence between the synthetic and original applications across instruction-set architectures, microarchitectures and compiler optimizations, and we point out the major sources of error in the benchmark synthesis process. We verified using software plagiarism detection tools that the synthetic benchmark clones indeed hide proprietary information from the original applications.

We argue that our framework can be used for several applications: distributing synthetic benchmarks as proxies for proprietary applications, drive architecture and compiler research and development, speed up simulation, model emerging and hard-to-setup workloads, and benchmark consolidation.

Conclusion. To facilitate benchmarking, we proposed different workload generation and reduction techniques. Code mutation transforms a proprietary application into a benchmark mutant that hides intellectual property while preserving performance characteristics. Sampling can be used to speed-up cycle-accurate simulation. Finally, high-level language benchmark synthesis generates short-running synthetic benchmarks that hide proprietary information.

Contents

List of Tables

List of Figures

List of Abbreviations

ASI	Architecture Starting Image
BBL	Basic Block
BLRL	Boundary Line Reuse Latency
CFO	Control Flow Operation
CPI	Cycles Per Instruction
D-Cache	Data Cache
DR	Dependence Ratio
ECF	Enforced Control Flow
HLL	High-Level Language
I-Cache	Instruction Cache
ILP	Instruction Level Parallelism
IR	Instruction Ratio
ISA	Instruction Set Architecture
KIPS	Kilo-Instructions Per Second
L1	Level-1 Cache
L2	Level-2 Cache
L3	Level-3 Cache
MA-CFO	Memory Access and Control Flow Operation
MHS	Memory Hierarchy State
MIPS	Million Instructions Per Second
MRRL	Memory Reference Reuse Latency
MSI	Microarchitecture Starting Image
NSL	No-State-Loss
PCA	Principal Component Analysis
RAW	Read After Write
RISC	Reduced Instruction Set Computer
ROB	Reorder Buffer
SFGL	Statistical Flow Graph with Loop information
SIR	Static Instruction Ratio
WDR	Weighted Dependence Ratio

WIR	Weighted Instruction Ratio

Chapter 1

Introduction

The beginning is the most important part of the work.
Plato

According to Moore's law [Moore, 1998], the number of transistors that can be integrated on a chip doubles approximately every two years. This is because of technological progress in the miniaturization of silicon transistors, i.e., transistors get smaller and cheaper. Computer architects and engineers exploit this empirical law in their quest for ever better performing microarchitectures. Over the years, microprocessors have evolved from scalar in-order execution processors to complex superscalar out-of-order processors. These processors issue multiple instructions per cycle, are deeply pipelined, have a deep memory hierarchy and employ branch prediction [Smith and Sohi, 1995]. Recently, there has been a shift towards multi-core and many-core processors [Olukotun et al., 1996].

This ever-increasing complexity leads to an increased design time of new microprocessors [Agarwal et al., 2000], and therefore an increased time-to-market. For contemporary designs, the design process can take as long as seven years [Mukherjee et al., 2002]. A large part of this time is spent on design space exploration or evaluating the performance impact of different design choices. Hence, designing new microprocessors and evaluating microprocessor performance are major challenges in computer architecture research and development [Skadron et al., 2003].

The growing microarchitectural complexity necessitates the use of benchmark programs to explore the design space of a microprocessor and to quantify the performance of a computer system. Benchmarks are

programs to assess the performance of a computer system. Computer architects and engineers quantify performance by running these programs and by timing their execution times. In the early design phases when the actual hardware is not yet available, benchmarks are used as input to an architectural simulator that models a new microarchitecture. The typical output of such a simulator is the number of cycles that are required to execute the considered benchmark, along with several statistics to help the architect understand how the different components of the simulated system behave, e.g., the number of instruction cache misses, the branch prediction accuracy, etc. Benchmarking is also useful when a computer system becomes available, e.g., to identify possible improvements that can be implemented in future designs, or to compare computer systems for guiding purchasing decisions.

Programs ranging from hand-coded synthetic benchmarks, micro-benchmarks, kernels, to application benchmarks have been employed as benchmarks for the performance evaluation of computer architectures. Synthetic benchmarks are programs that impose a desired workload on (an individual component of) a computer system, e.g., the microprocessor. They are usually developed based on statistics from one or more applications. Whetstone [Curnow and Wichmann, 1976] and Dhrystone [Weicker, 1984] are two well-known synthetic benchmarks that have been hand-coded in 1972 and 1984, respectively.

Micro-benchmarks [Sreenivasan and Kleinman, 1974] [Williams, 1977] [Wong and Morris, 1988] [Black and Shen, 1998] [Desikan et al., 2001] and kernels [McMahon, 1986] are programs designed to measure the performance on a very small and specific piece of code. The main difference between micro-benchmarks and kernels is that kernels are typically derived from real-world programs whereas micro-benchmarks are typically coded from scratch.

The caveat with these hand-coded synthetic benchmarks, micro-benchmarks and kernels is that they are not representative for real workloads. As a result, researchers rely heavily on application benchmarks, i.e., benchmarks that run real-world programs, to assess computer performance [Hennessy and Patterson, 2003]. Recently, there has been an emergence of these application benchmarks, e.g., SPEC CPU [Henning, 2000] [Henning, 2006], SPECmail, SPECweb, EEMBC benchmarks, TPC benchmarks [Pöss and Floyd, 2000], etc.

The availability of application benchmarks has standardized the process of performance comparison. However, two major aspects of

this benchmarking process demand close attention of computer architects and engineers: experimental design and data analysis. Experimental design refers to setting up the experiment, which involves benchmark design and selection, simulator development, etc. Data analysis refers to processing performance data after the experiment is run. The focus of this dissertation is on the former, i.e., the design of benchmarks to drive performance experiments in computer systems research and development.

1.1 Key challenges in benchmark design

The design of a benchmark is of paramount importance in systems research and development, and it involves several important challenges. In this section, we detail these challenges in benchmark design, which also gave rise to this research work.

1. Benchmarks should be representative of the (future) applications that are expected to run on the target computer system. A benchmark suite that ill-represents a target domain may eventually lead to a suboptimal design. However, it is not always possible to compose a representative benchmark suite for three reasons:

   (a) Industry-standard benchmarks are typically derived from open-source programs [Henning, 2000] [Henning, 2006]. The limitation is that these open-source benchmarks may not be truly representative of the real-world applications of interest. One solution may be to use real-world applications instead of the industry-standard benchmarks. However, in many cases, real-world applications cannot be distributed to third parties because of their proprietary nature.

   (b) Existing benchmarks are often outdated. The reason is twofold: (i) developing new benchmark suites and upgrading existing benchmark suites is extremely time-consuming and thus very expensive, and (ii) the application space is constantly evolving. As a result, the benchmark development process may lag the emergence of new applications.

   (c) Another problem arises when the software that is expected to run on a future computer system is still under construction. In this case, computer architects and engineers have to

rely on benchmarks that are modeled after existing applications that may be irrelevant by the time the product hits the market. One could argue that computer architects design tomorrow's systems with yesterday's benchmarks.

2. We also want a reduced benchmark. The redundancy[1] in a benchmark should be kept as small as possible to limit the amount of work that needs to be done during performance evaluation. This is especially important in the early design phases of a new microarchitecture, i.e., when computer architects and engineers rely on simulators for evaluating the performance of new ideas. These architectural simulators are extremely slow because of the complexity of the contemporary processors that they model. If we also take into account that (i) contemporary benchmarks execute hundreds of billions or even trillions of instructions, and (ii) during microarchitectural research a multitude of design alternatives need to be evaluated, we easily end up with months or even years of simulation time. This may stretch the time-to-market of newly designed microprocessors. To mitigate this problem, benchmarks should be short-running so that performance projections can be obtained through simulation in a reasonable amount of time.

3. A benchmark should enable both (micro)architecture and compiler research and development. Although existing benchmarks satisfy this requirement, this is typically not the case for workload reduction techniques that reduce the dynamic instruction count in order to address the simulation challenge. These techniques often operate on binaries and not on source code which eliminates their utility for compiler exploration and Instruction Set Architecture (ISA) exploration, e.g., when evaluating an architecture that includes ISA extensions.

In summary, we would like to have a set of benchmarks that is representative of existing and/or future applications, and that can be used for microarchitecture, architecture and compiler research. In addition, we would like to be able to reduce these representative benchmarks to limit the time spent during performance evaluation. In this research work, we propose three novel workload generation and reduction tech-

[1] This research work deals with intra-program redundancy, i.e., redundancy within a benchmark of a given benchmark suite.

niques that aim at addressing the aforementioned benchmarking challenges[2]. In the next section, we detail the specific contributions of this dissertation.

1.2 Contributions

We first propose a methodology to hide the proprietary information of contemporary applications to address the concern of distributing proprietary applications to third parties. Subsequently, we contribute to sampled simulation which addresses the long simulation time of contemporary benchmarks. Finally, we propose a workload generation technique that aims at fulfilling most of the aforementioned challenges and requirements. More precisely, this dissertation makes the following contributions to the workload generation for the performance evaluation of microprocessors.

Contribution 1: Code mutation

We propose code mutation which is a novel methodology that mutates a proprietary application to complicate reverse engineering so that it can be distributed as an application benchmark among third parties. These benchmark mutants hide the functional semantics of proprietary applications while exhibiting similar performance characteristics. The key idea of this methodology is to preserve the proprietary application's dynamic memory access and/or control flow behavior in the benchmark mutant while mutating the rest of the application code. We therefore exploit two key observations: (i) miss events (cache misses, branch mispredictions) have a dominant impact on performance on modern architectures, and (ii) many variables of contemporary applications exhibit invariant behavior at run time. We achieve this by computing dynamic program slices for memory access operations and/or control flow operations trimmed through constant value and branch profiles, and then mutating the instructions not appearing in these slices through binary rewriting.

[2]Note that the focus of this dissertation is on cpu-intensive single-threaded applications running on single-core processors. In Chapter 6, we discuss some of the challenges to extend the proposed techniques to multi-threaded applications running on multi-core processors.

We propose and empirically evaluate three approaches to code mutation for single-threaded applications. These approaches differ in how well they preserve the proprietary application's memory access and control flow behavior in the mutant. Obviously, there is a trade-off between information hiding and preserving the performance characteristics of the proprietary application. However, because code mutation is a top-down methodology — it mutates an existing (proprietary) application — the performance characteristics of the benchmark mutants are very similar to those of the proprietary applications, even for the most aggressive mutation approach.

We also introduce a set of novel metrics to quantify to what extent the functional meaning is hidden through code mutation. We report that code mutation is an effective approach that mutates up to 90% of the static binary and up to 50% of the dynamically executed instructions.

This work on code mutation is published in:

> Luk Van Ertvelde and Lieven Eeckhout, "Dispersing Proprietary Applications as Benchmarks through Code Mutation", In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008, 201-210.

Contribution 2: Cache state warmup for sampled processor simulation through NSL-BLRL

Sampled simulation addresses the long simulation times of contemporary benchmarks by simulating only a limited number of execution intervals from a complete benchmark execution, called sampling units, in a detailed manner. To make this approach feasible, the sampling units must provide an accurate representation of the entire program execution. Therefore, the sampling units must execute with an accurate picture of the processor state, i.e., the processor state should be close to the state should the whole dynamic instruction stream prior to the sampling unit be simulated in detail. The difference in state at the start of a sampling unit between sampled simulation and full simulation is often referred to in literature as the cold-start problem [Haskins and Skadron, 2003]. The structure that suffers the most from the cold-start problem is the memory hierarchy, i.e., the caches.

We propose NSL-BLRL, a novel cache warming approach that

builds on No-State-Loss (NSL) and Boundary Line Reuse Latency (BLRL) to efficiently approximate the cache state at the beginning of a sampling unit. The key idea of NSL-BLRL is to warm the cache hierarchy (at the beginning of a sampling unit) by loading a checkpoint that stores a truncated NSL stream. The NSL stream is the least-recently used (LRU) stream of memory references in the pre-sampling unit. This stream is then truncated using BLRL information, i.e., BLRL computes how many memory references are needed to accurately warm the cache hierarchy. Compared to BLRL, this approach yields simulation time speedups up to 1.4× under fast-forwarding and up to 14.9× under checkpointing, while being nearly as accurate. Also, NSL-BLRL is on average 30% more space-efficient than NSL for the same level of accuracy.

A discussion on this hybrid cache state warm-up approach for sampled processor simulation is published in:

> Luk Van Ertvelde, Filip Hellebaut, Lieven Eeckhout and Koen De Bosschere, "NSL-BLRL: Efficient Cache Warmup for Sampled Processor Simulation", In *Proceedings of the Annual Simulation Symposium (ANSS)*, 2006, 87-96.
>
> Luk Van Ertvelde, Filip Hellebaut and Lieven Eeckhout, "Accurate and Efficient Cache Warmup for Sampled Processor Simulation through NSL-BLRL", In *The Computer Journal*, Vol. 51, No. 2, 192-206, March 2008.

Contribution 3: High-level language benchmark synthesis

We propose a novel benchmark synthesis framework for generating short-running benchmarks that are representative of other applications without revealing proprietary information. The key novelty of our approach is that these synthetic benchmarks are generated in a High-Level programming Language (HLL), which enables both architecture and compiler research. In contrast to code mutation, benchmark synthesis is a bottom-up methodology, i.e., it generates synthetic benchmarks from collected program characteristics. This allows us to reduce the number of dynamically executed instructions.

The methodology consists of two key steps: (i) profiling a real-world proprietary application to measure its execution characteristics, and (ii) modeling these characteristics into a synthetic benchmark

clone. We introduce a novel structure to capture a program's control flow behavior in a statistical way: the Statistical Flow Graph with Loop information (SFGL). The SFGL enables our framework to generate function calls, (nested) loops and conditional control flow behavior in the synthetic benchmark. Prior work [Bell and John, 2005] [Bell et al., 2006] [Joshi et al., 2007] [Joshi et al., 2008a] [Joshi et al., 2008b] in benchmark synthesis instead generated a long sequence of basic blocks, but no loops nor function calls. In addition, it was limited to the generation of synthetic benchmarks at the assembly level.

We evaluate our framework on x86 and IA64 hardware using single-threaded workloads. We consider multiple hardware platforms with different instruction set architectures, microarchitectures, compilers and optimization levels. Prior work was limited to the evaluation through simulation, and/or considered Reduced Instruction Set Computer (RISC) ISAs. Finally, we use software plagiarism tools to demonstrate that our synthetic benchmarks do not reveal proprietary information.

A discussion on benchmark synthesis at the high-level language is published in:

> Luk Van Ertvelde and Lieven Eeckhout, "Benchmark Synthesis for Architecture and Compiler Exploration", In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2010, 106-116.

Contribution 4: Comparison of workload generation and reduction techniques

To guide the process of performance evaluation, we give an overview of other recently proposed workload generation and reduction techniques. We compare them with our techniques along the following criteria: whether they yield representative and short-running benchmarks, whether they can be used for both architecture and compiler exploration, and whether they hide proprietary information.

We conclude there is no clear winner, i.e., the different techniques represent different trade-offs. The trade-off between code mutation and benchmark synthesis is that benchmark synthesis yields synthetic benchmarks that may be less accurate and representative with respect to the real-world workloads compared to mutated binaries; however, it hides proprietary information more adequately and it yields shorter-

running benchmarks. Furthermore, high-level language benchmark synthesis seems to be one of the few techniques that aims at enabling compiler and ISA exploration.

This comparison on workload generation and reduction techniques is published in:

> Luk Van Ertvelde and Lieven Eeckhout, "Workload Reduction and Generation Techniques", In *IEEE Micro*, Nov/Dec 2010, Vol. 30, No.6.

1.3 Overview

This dissertation is organized as follows.

We present code mutation in Chapter 2. We discuss how we profile a proprietary application to collect its behavior characteristics, and how we use these characteristics to rewrite the proprietary application into a benchmark mutant. Also, we introduce a set of novel metrics for quantifying how well the functional meaning of a proprietary application is hidden by the benchmark mutant. Finally, we quantify how well the performance characteristics of the proprietary application are preserved in the mutant.

In Chapter 3, we briefly revisit sampled processor simulation. Subsequently, we propose NSL-BLRL for minimizing the cost associated with warming the cache hierarchy in sampled simulation.

Chapter 4 describes the benchmark synthesis framework for generating a synthetic benchmark in a high-level programming language from desired program characteristics. We detail the profiling step and the synthetic benchmark generation process. Finally, we demonstrate that we indeed can generate representative synthetic benchmarks that are shorter running than the workloads they are modeled after, while still hiding proprietary information.

In Chapter 5, we compare the techniques that we proposed in the previous chapters with other recently proposed workload generation and reduction techniques.

Chapter 6 concludes this dissertation with a summary and a discussion of future research directions.

Chapter 2

Code mutation

If I were two-faced, would I be wearing this one?
Abraham Lincoln

In this chapter, we propose code mutation, a novel methodology for constructing benchmarks that hide the functional semantics of proprietary applications while exhibiting similar performance characteristics. As such, these benchmark mutants can be distributed among third parties as proxies for proprietary applications for performance evaluation purposes.

After discussing the general idea of code mutation, we extensively describe the code mutation framework and we motivate a number of high-level design decisions. We explore several code mutation approaches and evaluate for each approach how well the performance characteristics of the proprietary application are preserved in the mutant, and how well the functional meaning of the proprietary application is hidden by the mutant. To quantify this information hiding, we propose a set of novel metrics.

2.1 Introduction

2.1.1 Proprietary nature of real-world applications

Benchmarking is the key tool for assessing computer system performance. One use of benchmarking is to drive the design process of next-generation processors. Another use of benchmarking is to compare computer systems for guiding purchasing decisions. In order to enable a fair evaluation of computer system performance, organizations such as EEMBC, TPC and SPEC standardize the benchmarking process, and

for this purpose these organizations provide industry-standard benchmark suites. These benchmark suites cover different application domains, as illustrated in Table 2.1.

Table 2.1: Examples of modern benchmarks suites.

Application domain	Example benchmark suites
bio-informatics	BioPerf [Bader et al., 2005]
databases	TPC [Pöss and Floyd, 2000]
general-purpose	SPEC CPU [Henning, 2000] [Henning, 2006]
graphics	SPECgpc
Java	DaCapo [Blackburn et al., 2006], SPECjbb, SPECjvm
multimedia	MiBench [Guthaus et al., 2001], MediaBench [Lee et al., 1997]
transactional memory	STAMP [Cao Minh et al., 2008]

The limitation of these industry-standard benchmarks is that they are typically derived from open-source programs [Henning, 2000] [Skadron et al., 2003] which may not be representative of real-world applications [Maynard et al., 1994] and may be very different from real-world applications of interest. In addition, available benchmark suites are often outdated because the application space is constantly evolving and developing new benchmark suites is extremely time-consuming (and thus very expensive). An alternative would be to use real-world applications; however, companies are not willing to release their applications because of intellectual property issues. For example, a cell phone company may not be willing to share its next-generation phone software with a processor vendor for driving the processor design process. This may eventually lead to a processor design that provides suboptimal performance for the target phone software. In other words, it would be in the industry's interest to be able to distribute real-world applications so that the computer systems are designed to provide good performance for these applications. We propose code mutation to address this concern of distributing proprietary applications to third parties [Van Ertvelde and Eeckhout, 2008].

2.1.2 General idea of code mutation

The objective of code mutation is to modify a proprietary application into a benchmark mutant that meets two conditions: (i) the functional

semantics of the proprietary application cannot be revealed, or, at least, are very hard to reveal through reverse engineering of the mutant, and (ii) the performance characteristics of the mutant resemble those of the proprietary application well so that the mutant can serve as a proxy for the proprietary application during benchmarking experiments. The key idea of our methodology is to approximate the performance characteristics of a proprietary application by retaining memory access and/or control flow behavior while mutating the remaining instructions. To achieve this, we first compute program slices for memory access operations and/or control flow operations. We then trim these slices using constant value and branch profiles — the reason for trimming these slices is to make more instructions eligible for code mutation. Finally, we mutate the instructions not appearing in these slices through binary rewriting. The end result is a benchmark mutant that can be distributed to third parties for benchmarking purposes, which may ultimately lead to more realistic performance assessments with respect to real-world applications.

2.2 Design options

Before describing our code mutation framework in detail, we discuss the high-level design choices we faced when developing our code mutation framework. We also highlight the key difference between code mutation and benchmark synthesis (we will discuss benchmark synthesis in Chapter 4).

Trace mutation versus benchmark mutation. A first design option is to distribute a mutant in the form of a trace or a benchmark. A trace, or a sequence of dynamically executed instructions, is harder to reverse engineer than a benchmark. Techniques borrowed from statistical simulation[1] to probabilistically instantiate program characteristics in a synthetic trace [Oskin et al., 2000] [Nussbaum and Smith, 2001] [Eeckhout et al., 2004], or coalescing representative trace fragments [Conte et al., 1996] [Iyengar et al., 1996] [Sherwood et al., 2002] [Wunderlich et al., 2003] [Ringenberg et al., 2005] could be used to mutate the original trace so that the functional meaning is hidden. A major limitation for a trace mutant is that it cannot be executed on hardware nor on execution-

[1] We will further detail statistical simulation at the end of Chapter 4.

driven simulators, which is current practice in simulation. We therefore choose to create benchmark mutants instead of trace mutants.

One input. A second design option is to mutate the proprietary application so that the resulting mutant can still take different inputs at run time, or to intermingle the input with the application when creating the mutant. We choose for the latter option for two reasons: (i) intermingling the input and the application enables more aggressive code mutations, and (ii) it prevents a malicious individual to try reverse engineer the proprietary application by applying different inputs to the mutant. The disadvantage of this approach is that a benchmark mutant needs to be generated for each particular input set.

Binary level versus source code level mutation. Mutation can be applied at the binary level or at the program source code level. The advantage of source code level mutation is that it is easier to port across platforms, and it enables distributing mutants for compiler research and development (which is an important benchmarking challenge, as described in the introduction of this dissertation). Nevertheless, making sure that the performance characteristics of the mutant correspond to those of the proprietary application is a challenging task because the compiler may affect the performance characteristics differently for the benchmark mutant than for the proprietary application. We first choose to generate benchmark mutants at the binary level; in Chapter 4, we investigate the generation of benchmarks in a high-level programming language to enable compiler research and development as well.

Bottom-up versus top-down benchmark mutation. Code mutation is a top-down approach that mutates an existing application to hide proprietary information as much as possible while retaining performance behavior so that proprietary applications can be shared as benchmark mutants. A different approach to distribute proprietary applications as benchmarks is to generate a synthetic benchmark clone[2] from desired program characteristics [Bell and John, 2005] [Joshi et al., 2006b]. The research challenge of this bottom-up approach is to identify the

[2] A similar distinction between benchmark cloning and benchmark mutation is made in circuit synthesis, an area related to benchmark synthesis [Verplaetse et al., 2000].

key program characteristics that when modeled in the synthetic benchmark, resemble the original application well. Intuitively, the advantage of synthetic benchmark generation is that hiding functional meaning is easier, whereas code mutation eases achieving the goal of preserving the behavioral characteristics of the proprietary workload. For this reason, we investigate benchmark synthesis as well, see Chapter 4. The trade-off between both approaches is discussed in Chapter 5.

Existing program obfuscation techniques. Code obfuscation [Collberg et al., 1997] is a growing research topic of interest which converts a program into an equivalent program that is more difficult to understand and reverse engineer. Although benchmark mutation has some properties in common with code obfuscation, there are two fundamental differences. First, code obfuscation requires that the obfuscated program version is functionally equivalent to the original program, and produces the same output. A mutant on the other hand does not require to be functionally equivalent, and may even produce meaningless output. Second, a mutant should exhibit similar behavioral characteristics as the proprietary application. An obfuscated program version on the other hand does not have this requirement, and as a matter of fact, many code obfuscation transformations change the behavioral execution characteristics through control flow and data flow transformations and by consequence introduce significant run time overheads, see for example [Ge et al., 2005]. These differences call for a completely different approach for code mutation compared to code obfuscation.

2.3 Code mutation framework

The general framework for code mutation is depicted in Figure 2.1. First, we profile the proprietary program's execution, i.e., we run the proprietary application along with a proprietary input within an instrumentation framework for collecting a so-called 'execution profile'. The execution profile is then used to transform the application code into a mutant through binary rewriting. The mutant can then be shared among third party industry vendors, as well as between industry and academia.

We started from the observation that performance on contemporary superscalar processors is primarily determined by miss events [Karkhanis and Smith, 2004], i.e., branch mispredictions and cache

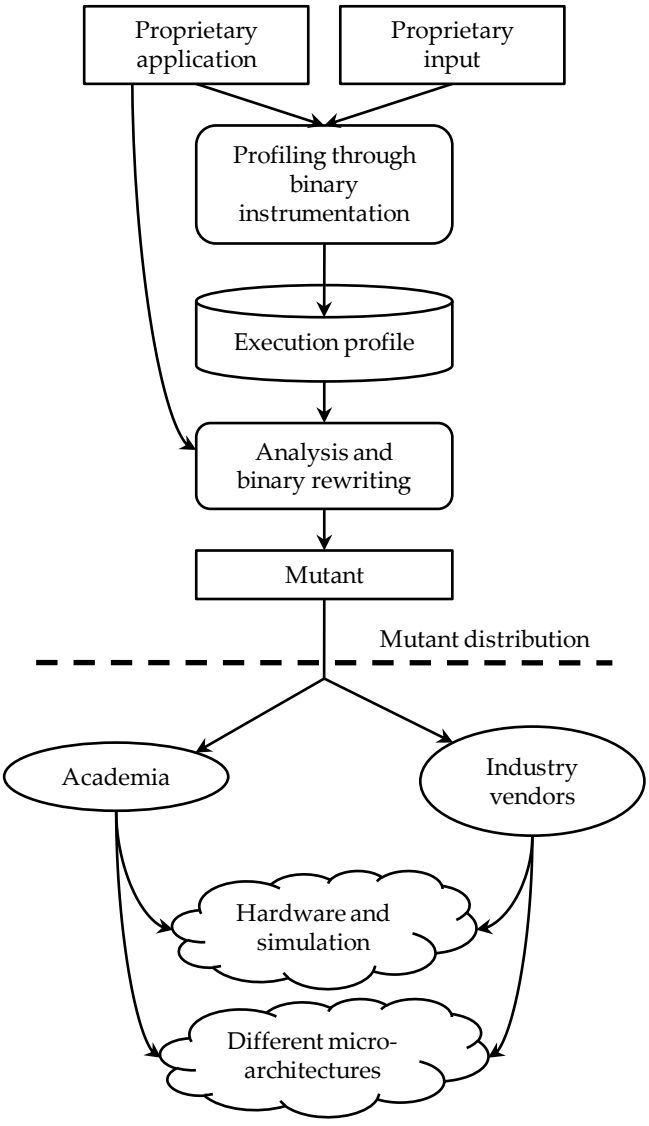

Figure 2.1: The code mutation framework.

misses, and to a lesser extent by inter-operation dependencies and instruction types; inter-operation dependencies and instruction execution latencies are typically hidden by superscalar instruction processing. This observation suggests that the mutant, in order to exhibit similar behavioral characteristics as the proprietary application, should mimic the branch and memory access behavior well without worrying too much about inter-operation dependencies and instruction types. In order to do so, we determine all operations that affect the program's branch and/or memory access behavior; we do this through dynamic program slicing (see Section 2.3.2). We retain the operations appearing in these slices unchanged in the mutant, and all other operations in the program are overwritten to hide the proprietary application's functional meaning.

In Sections 2.3.1 through 2.3.3, we provide details on the three key steps in our framework: (i) profiling the (real-world) proprietary application to collect execution characteristics, (ii) program analysis using this execution profile to mark instructions that can be mutated, and (iii) mutating the proprietary workload into a benchmark mutant through binary rewriting.

2.3.1 Collecting the execution profile

The execution profile consists of three main program execution properties: (i) the inter-operation dependency profile, (ii) the constant value profile, and (iii) the branch profile. The execution profile will be used in the next step by the slicing algorithm for determining which operations affect the branch and/or memory access behavior.

This execution profile can be collected using a binary instrumentation tool such as ATOM [Srivastava and Eustace, 1994] or Pin [Luk et al., 2005]. We use Pin in our framework. Dynamic binary instrumentation using Pin adds instrumentation code to the application code as it runs that collects the program execution properties of interest.

Inter-operation dependency profile

The inter-operation dependency profile captures the data dependencies between instructions. Specifically, it computes the read-after-write (RAW) dependencies between instructions through both registers and memory. The inter-operation dependency profile then enumerates all

the static instructions that a static instruction depends upon (at least once) at run time.

For example, consider the example given in Figure 2.2, where instruction `d` consumes registers `r3` and `r6`; instruction `a` produces `r3` and both instructions `b` and `c` produce `r6`. If it turns out that the path shown through the thick black line (A-B-D) is always executed, i.e., basic block C is never executed, then only instructions `a` and `b` will appear in the inter-operation dependency profile for instruction `d`. Instruction `c` will not appear in the dependency profile because basic block C is never executed in this particular execution of the program. If, in contrast, basic block C would be executed at least once, then `a`, `b` and `c` would appear in the dependency profile for instruction `d`.

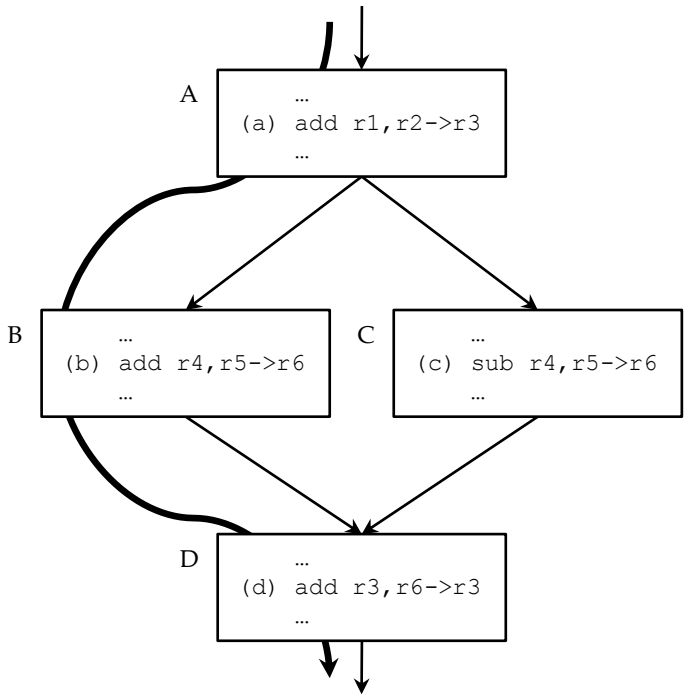

Figure 2.2: An example illustrating the inter-operation dependency profile: if basic block C is never executed, only instructions `a` and `b` will appear in the inter-operation dependency profile for instruction `d`.

Constant value profile

For each instruction we profile whether the register values that it consumes and the register values that it produces are constant over the entire program execution. In other words, for each instruction, we keep track of the register values that it consumes and produces during profiling, and store the constant value register operands in the execution profile.

Value locality [Lipasti et al., 1996] [Lipasti and Shen, 1996], value profiling [Calder et al., 1997], and its applications have received great research interest over the past decade. Various authors have reported that a substantial fraction of all variables produced by a program at run time is invariant or semi-invariant. For example, Lipasti and Shen [1996] show for a collection of integer benchmarks that on average 49% of the instructions write the same value as they did the last time. Calder et al. [1997] explore the invariant behavior of values for loads, parameters, and register operations. There are several ways of leveraging invariant (and semi-invariant or predictable) data values such as hardware value prediction [Lipasti and Shen, 1996], code specialization [Calder et al., 1999], partial evaluation, and adaptive and dynamic optimization [Auslander et al., 1996]. We will use the constant value profiles to trim the program slices as we will explain in Section 2.3.2.

Branch profile

The branch profile captures a number of profiles concerning a program's control flow behavior:

- We store the branch direction in case a conditional branch is always taken or always not-taken.
- We store the branch target in case an indirect jump always branches to the same target address.
- In case of an unconditional jump, we choose a condition flag that is constant at the jump point across the entire program execution.
- And finally, we also capture the taken/not-taken sequence for infrequently executed conditional branches, i.e., in case the branch is executed less than 32 times during the entire program execution.

Non-determinism

Non-deterministic system calls complicate the computation of the constant value and branch profiles: a variable in one program run may have a different value in another program run with the same input because of system call effects. We take a pragmatic solution and run each program multiple times and then subsequently compute the constant value profiles across these program runs. A more elegant solution may be to record/replay system effects as done in pinSEL [Narayanasamy et al., 2006].

2.3.2 Program analysis

The second step in our framework is to use the execution profile to mark the instructions that can be mutated. To achieve this, we first compute the operations that affect the branch and memory access behavior of a program execution. Computing these operations is done through program slicing — program slicing itself is supported by the inter-operation dependency profile. We then trim the program slices using the constant value profile and the branch profile, and we mark all the instructions not part of a slice as eligible for code mutation.

Program slicing

As alluded to before, we use *program slicing* [Weiser, 1981] [Tip, 1995], which is a powerful technique for tracking chains of dependencies between operations in a program. Program slicing is found useful for various purposes in software development [Calder et al., 1997], testing and debugging [Weiser, 1982] [Gupta et al., 2005], as well as in optimizing performance through identifying critical operations [Zilles and Sohi, 2000]. A *program slice* consists of the instructions that (potentially) affect the computation of interest, referred to as the *slicing criterion*. In this work we consider *backward* slices, or the sequence of instructions leading to the slicing criterion. The backward slice, or slice for short, can be computed through a backward traversal of the program starting at the slicing criterion. An important distinction is to be made between a *static* and a *dynamic* slice [Agrawal and Horgan, 1990]. The former does not make any assumptions about the program's input whereas the latter relies on a specific test case.

Our framework uses dynamic slicing because we have a specific

input available, and because dynamic slices are typically thinner, i.e., contain fewer instructions, than static slices. This enables us to more aggressively apply code mutation. We use an *imprecise* dynamic slicing algorithm because of the high computational complexity both in time and space of precise dynamic slicing [Zhang et al., 2005], especially for computing many slices for long-running applications. The slices produced through imprecise slicing are less accurate than through precise slicing, nevertheless they are conservative meaning that they are a superset of precise slices. As will turn out, this means that a precise dynamic slice is a subset of an imprecise dynamic slice.

We use Algorithm II as described by Agrawal and Horgan [1990] and Zhang et al. [2005]. This algorithm[3] starts from the slicing criterion and recursively builds the backward slice using the inter-operation dependency profile: starting from the dependency profile for the slicing criterion, it recursively computes prior dependencies. The computational cost for this imprecise slicing algorithm is independent of the number of slices that need to be computed [Zhang et al., 2005]. This is an important benefit for our purpose since we compute slices for all control flow operations and/or data memory accesses, as explained in the next section.

Figure 2.3 illustrates the dynamic slicing algorithm through an example, and how we apply this algorithm to calculate assembly level slices. The thick black line represents the dynamic sequence of basic blocks for a particular run of a program: A-B-D-E-H-A-C-D-F. Assume we want to compute the slice for branch instruction (h2). The imprecise dynamic slice comprises instructions {`a`, `b`, `c`, `d`, `e`, `h1`}, and is computed as follows. Instruction `h2` depends on instruction `h1` through register `r3`. Instruction `h1` in its turn depends on instruction `d` through register `r3` as well as on instruction `e` through memory location `mem1` — `h1` does not depend on `f` and `g` in this particular execution. Instruction `d` in its turn depends on `a`, `b` and `c` as these three operations appear in the inter-operation dependency profile for `d`. Note however that although `d` does not depend on `c` along the A-B-D-E-H execution path, it still appears in the dynamic slice for `h2` — `c` appears in the inter-operation dependency profile for `d` just because `c` produces a value that is consumed by `d` along the A-C-D-F execution path. This is exactly where the imprecise dynamic slice differs from the precise dy-

[3]Note however that we apply this algorithm to calculate slices at the assembly level instead of at the level of a high-level programming language.

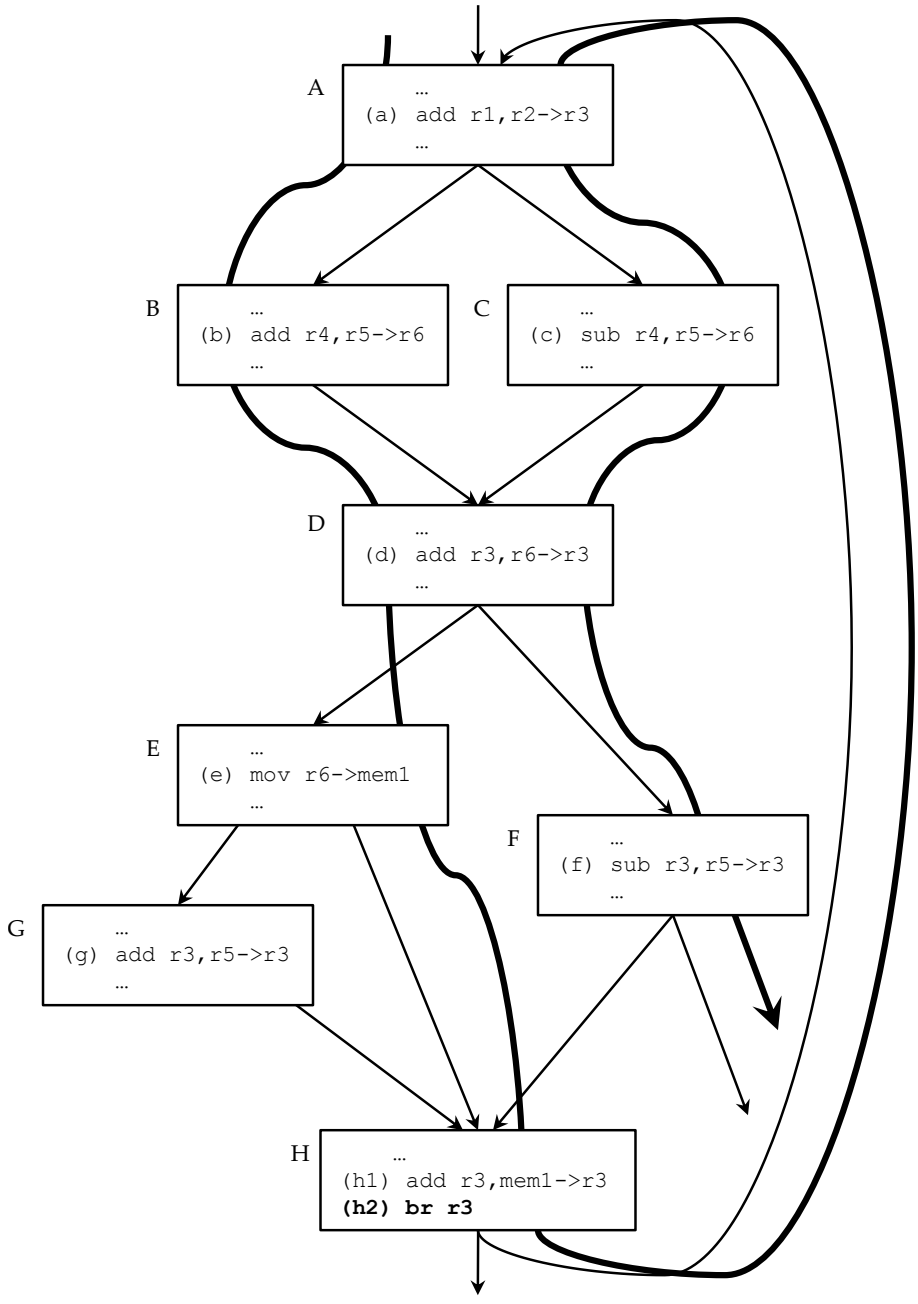

Figure 2.3: An example illustrating program slicing at the assembly level. The imprecise dynamic slice of branch instruction `h2` equals {`a`, `b`, `c`, `d`, `e`, `h1`}.

namic slice: the precise dynamic slice does not contain `c` and equals {`a`, `b`, `d`, `e`, `h1`}. Although the imprecise dynamic slice is slightly less accurate than the precise dynamic slice, it is smaller than the static slice for `h2` which comprises instructions {`a`, `b`, `c`, `d`, `e`, `f`, `g`, `h1`}: instructions `f` and `g` also appear in the slice although there is no dependency between `f/g` and `h1` at run time.

The constant value and branch profiles help trimming both the number of slices as well as the size of the slices that need to be computed. Specifically, we do not need to compute slices for branches that are either always taken or always not-taken. Also, the recursive backward dependency tracking for computing the slices stops upon a constant value. For example, if the value for register `r3` in instruction `d` is invariant, the trimmed dynamic slice of instruction `h2` will no longer include instructions `a`, `b` and `c`.

Code marking

In our evaluation, we consider three scenarios with different slicing criteria.

- **Memory Access and Control Flow Operation (MA-CFO) slicing:** The first scenario computes slices for all control flow operations as well as for all effective data addresses generated through loads and stores. This scenario will ensure that the control flow and memory access behavior of the mutant will be identical to the proprietary application.
- **Control Flow Operation (CFO) slicing:** The second scenario only computes slices for all control flow operations. This criterion will be less accurate than the former because it does not compute slices for data memory accesses. This has the potential benefit of enabling more aggressive code mutation at the potential cost of the mutant being less representative of the proprietary application.
- **CFO + Enforced Control Flow (ECF) slicing:** The third scenario computes slices for all control flow operations except conditional branches that are executed less than 32 times over the entire program execution — we detail this approach at the end of the following subsection.

Once the slices are computed, all the instructions not part of a trimmed slice are marked as eligible for code mutation. Also, all instruction operands (input as well as output operands) that hold constant values at run time are marked as such.

2.3.3 Binary rewriting

The final step in our code mutation framework is to perform the actual code mutation. We employ binary rewriting to mutate the proprietary application into a benchmark mutant; we use Diablo [De Bus et al., 2003] as our binary rewriting tool. Applying mutation through binary rewriting poses some challenges in terms of preserving the code layout. Rewriting instructions may change the code layout and may thereby affect the instruction cache performance. We therefore strive at keeping the basic block size the same before and after mutation.

Control flow mutations

As mentioned before, we do not compute slices for branches that have a constant branch target. Instead, we mutate those branches to complicate the understanding of the mutant. To do so, we use an opaque variable or predicate [Collberg et al., 1997]. An opaque variable is a variable that has some property that is known a priori to the code mutator, but which is difficult for a malicious person to deduce. In our work, we use as the opaque variable a condition flag that has a constant value at the branch point during the course of the program execution but which is different from the condition flag in the original program. We mutate conditional branches that are either always taken or always not-taken, indirect jumps with a constant branch target, and we also convert unconditional branches into conditional branches. Conditional branches that jump based on an opaque condition flag do not alter the execution flow of the mutant but complicate the understanding of the mutant binary. In addition, control flow edges that are never taken are altered in the mutant.

Rewriting code and breaking data dependencies

We rewrite the marked instructions — including the unexecuted code — by randomly reassigning an instruction type, and register input and output operands. This random reassignment assures that there is no

one-to-one mapping of instruction types and operands which complicates reverse engineering. Nevertheless, we ensure that the dynamic instruction mix in the mutant — the relative occurrence of instruction types — is similar to that in the proprietary application so that the run time behavior characteristics of the mutant match those of the proprietary application. Likewise, we generate inter-operation data dependencies in such a way that the distribution of inter-operation dependencies of the mutant is similar to the one of the proprietary application in order to preserve the amount of ILP (Instruction-Level Parallelism) in the mutant. The instruction operands that are marked as holding constant values are replaced by immediate constants. By doing so, we break inter-operation dependencies making it harder to understand the proprietary application. For the output register operands, we use non-live register operands to make sure the inserted code mutations do not affect the execution flow of the mutant.

Example

We now illustrate code mutation that preserves the control flow behavior on a simple example that computes the factorial of 7. Figure 2.4 shows the original factorial function and Figure 2.5 shows the code after code mutation; the instructions shown in bold in Figure 2.5 are overwritten.

The input to the function, which is '7', is held in register `eax`. Basic block A checks whether the input is larger than 12. If yes, error handling code is executed in E; if no, the factorial is computed and printed in B, C, and D. The branch instruction in A is a conditional branch that is never taken for the given input '7': we overwrite this branch as well as the `cmp` instruction in its slice. E is never executed, and by consequence, we can completely overwrite E. Also the conditional branch in B is never taken — the input differs from '1' — and we thus overwrite the branch and the instructions in its slice. In C, both `eax` and `edx` from the `cmp` instruction appear in the slice for the conditional branch at the end of C. However, the value for `eax` is invariant for this particular execution, and we thus overwrite the `eax` argument by its constant value which is 7. For register `edx`, the slice includes the `cmp` and `inc` instructions in C and the `mov` instruction in B; these instructions thus remain unchanged in the mutant. The values in `eax` and `ebx` in D are constant, and are overwritten by constant values. As a result of that, we can overwrite the `imul` instruction in C. The end result of code muta-

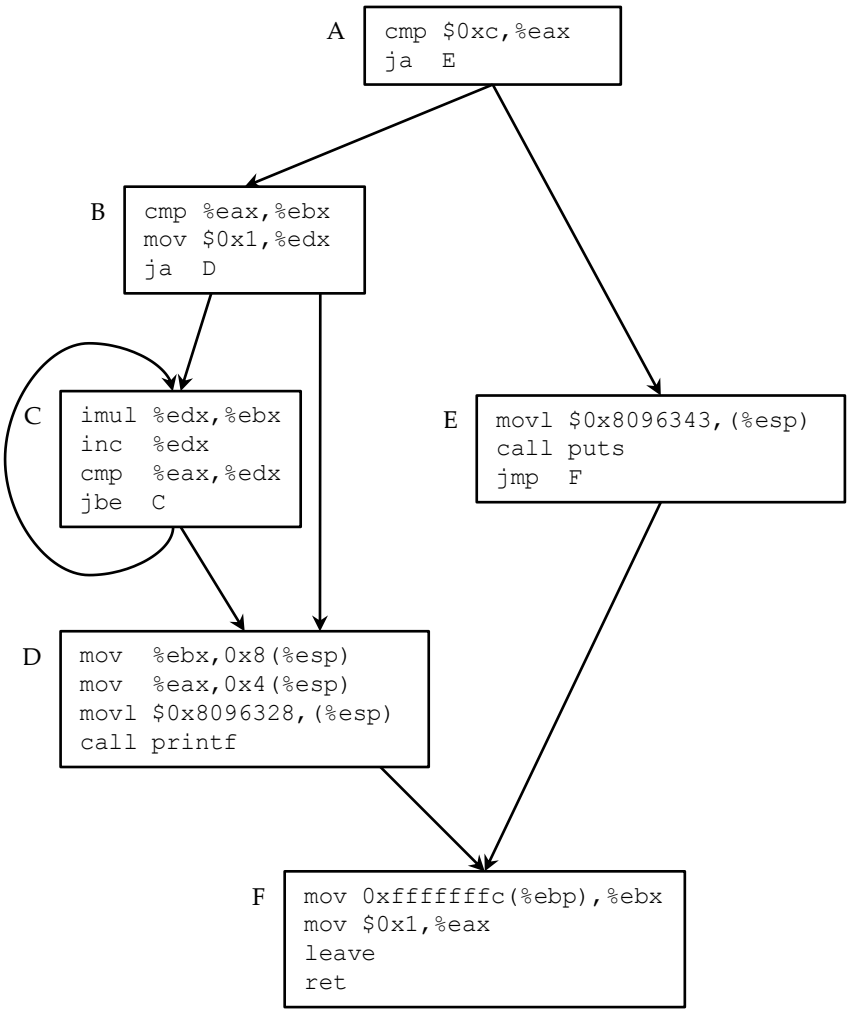

Figure 2.4: Factorial function before code mutation.

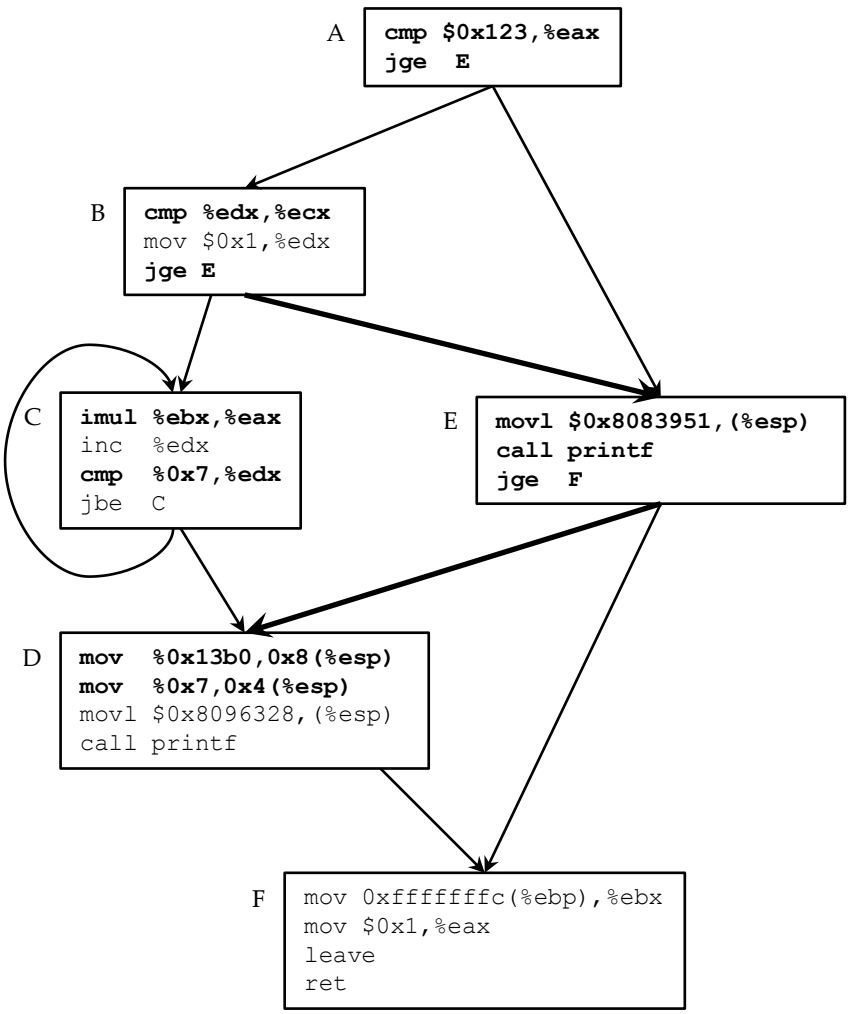

Figure 2.5: Factorial function (with input '7') after code mutation. Instructions that are mutated are shown in bold.

tion is a mutant, shown in Figure 2.5, that looks fairly different from the original application shown in Figure 2.4. It will be very hard to reveal the functional meaning of the original application from its mutant.

Infrequent branches

As already alluded to, we do not compute slices for infrequent branches, in our case, conditional branches that are executed less than 32 times over the entire program execution. For those branch instructions we record the branch taken/not-taken sequence in the branch profiles (as mentioned above), and replay this branch sequence in the mutant at run time. We refer to this transformation as 'Enforced Control Flow' (ECF). The following code snippet, a basic block from the `crafty` benchmark, illustrates ECF:

Original application before ECF:

```
(1) and %edx,%eax
(2) mov 0x154(%esp),%ebx
(3) mov %eax,0x150(%esp)
(4) or  0x150(%esp),%ebx
(5) jne 0x807026f
```

Benchmark mutant after ECF:

```
(a) xor %ebx,%eax
(b) mov 0x154(%esp),%ecx
(c) mov %ebx,0x150(%esp)
(d) mov 0x80637e9,%ecx
(e) shl %ecx
(f) mov %ecx,0x80637e9
(g) js  0x80639f6
```

To enforce the taken/not-taken behavior for the conditional branch, we first load the branch sequence from memory, see instruction `d` in the benchmark mutant — the branch sequence is stored in memory location `0x80637e9`. We then shift left this branch sequence, and subsequently store it back to memory, see instructions `e` and `f`. Finally, the sign bit determines the branch direction (see instruction `g`). An important benefit of ECF is that it increases the number of instructions eligible for code mutation. For example, the instructions `1` through `3` are mutated because of not having to retain these instructions in the slice leading up to the conditional branch.

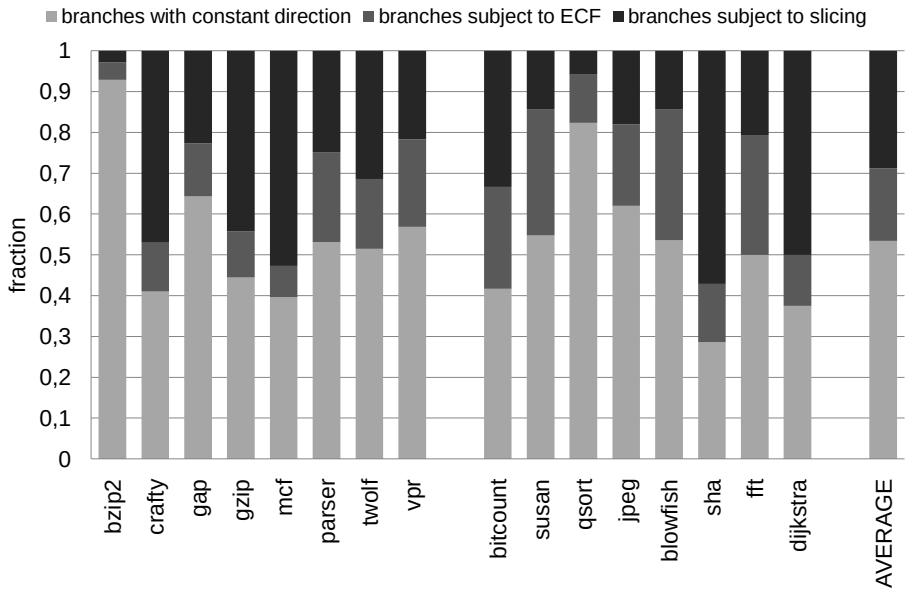

Figure 2.6: Fraction of the branches in the binary (i) with constant taken/not-taken direction, (ii) that are subject to ECF (Enforced Control Flow), and (iii) that are subject to slicing.

Figure 2.6 quantifies for what fraction of branches in the static binary we can apply ECF — we will describe the experimental setup later in Section 2.5. We can apply ECF to approximately 18% of the static branches. Together with the on average 54% of the branches that are always taken or not-taken, this means, that on average, we only need to compute slices for 28% of the static branches.

2.4 Quantifying Mutation Efficacy

There are two issues when quantifying the efficacy of code mutation. The first one concerns with how well the performance characteristics of the proprietary application are preserved in the mutant. This is straightforward to do: this can be done by comparing performance numbers of the mutant against the proprietary application across a number of microarchitectures and hardware platforms. The second issue is much more challenging and concerns with how well the functional meaning of the proprietary application is hidden by the mutant.

An industry vendor will only release a proprietary application as a benchmark mutant based on a thorough efficacy evaluation. This section discusses metrics for quantifying how well the functional meaning of a proprietary application is hidden by the mutant.

There exist a vast body of work on software complexity metrics which typically quantify various textual properties of the source code into a complexity measure. Code obfuscation uses some of those metrics to quantify the efficacy of code transformations [Collberg et al., 1997]. These metrics relate to code size, data flow complexity, control flow complexity, data structure complexity, etc. These metrics only partially achieve what we want a mutation efficacy metric to measure. Recall that the aim of code obfuscation is to complicate the understanding of the proprietary application; however, the obfuscated program is still functionally equivalent to the original application — code obfuscation just makes it harder to grasp the proprietary information. Code mutation goes one step further in the sense that a mutant removes information from the proprietary application through binary rewriting, i.e., the mutant is no longer functionally equivalent to the proprietary application.

These considerations call for metrics specifically targeted towards quantifying how well code mutation hides the proprietary information. We therefore develop a set of metrics to quantify mutation efficacy. The first three metrics quantify the number of instructions that are mutated, i.e., the fraction instructions that have been rewritten. The next two count the number of data dependencies that are broken in the mutant with respect to the proprietary application — hiding data dependencies and introducing artificial data dependencies complicate the understanding of the mutant. In this work, when reporting these metrics, we only report about the application, not the libraries — the reason is to stress code mutation in the evaluation because most library code is executed infrequently (if at all).

- **Static Instruction Ratio (SIR):** The SIR computes the ratio of the number of instructions in the binary that are mutated relative to the total number of instructions in the binary.
- **Static Executed Instruction Ratio (IR):** The SEIR computes the ratio of the number of instructions in the binary that are mutated and executed at least once, relative to the number of instructions in the binary that are executed at least once.

- **Weighted Instruction Ratio (WIR):** The WIR computes the ratio of the number of instructions that are mutated, weighted by their execution frequency relative to the total dynamic instruction count. In other words, the WIR computes the fraction mutated instructions executed relative to the dynamic instruction count.
- **Dependence Ratio (DR):** The DR metric computes the fraction inter-operation data dependencies that appear at least once at run time and that are broken in the mutant.
- **Weighted Dependence Ratio (WDR):** The WDR metric weights the DR metric with the execution frequency for each of the dependencies.

The first metric quantifies the efficacy of code mutation for making *static reverse engineering* hard, i.e., reverse engineering of the proprietary application by inspecting the binary of the benchmark mutant. The other four metrics quantify the efficacy for making *dynamic reverse engineering* hard, i.e., reverse engineering by inspecting the dynamic execution of the mutant.

2.5 Experimental setup

In our evaluation we consider a number of general-purpose SPEC CPU2000 [Henning, 2000] benchmarks as well as a number of benchmarks from the embedded MiBench benchmark suite [Guthaus et al., 2001]. The benchmarks[4] are shown in Table 2.2 and Table 2.3 along with their inputs. For the SPEC CPU2000 benchmarks, we use MinneSPEC [KleinOsowski and Lilja, 2002] inputs in order to limit the simulation time of complete benchmark executions. For MiBench, we consider the largest input available. All the benchmarks are compiled on an x86 platform (Intel Pentium 4 running Linux) using `gcc` compiler version 3.2.2 with optimization level `-O3`; all binaries are statically compiled.

The baseline processor configuration is shown in Table 2.4. We model a 4-wide superscalar out-of-order processor with a three-level cache hierarchy. The simulations are done using PTLsim [Yourst, 2007], an execution-driven x86 superscalar processor simulator.

[4]We were unable to include all the SPEC CPU2000 integer benchmarks because of difficulties in inter-operating the various tools (Pin, Diablo and PTLsim) in our experimental setup.

Table 2.2: The SPEC CPU2000 benchmarks used for evaluating code mutation.

Benchmark	Description	Input
bzip2	compression	lgred.source
crafty	game playing: chess	lgred
gap	group theory, interpreter	lgred
gzip	compression	smred.log
mcf	combinatorial optimization	lgred
parser	word processing	lgred
twolf	place and route simulator	lgred
vpr	FPGA circuit placement and routing	small.arch

Table 2.3: The MiBench benchmarks used for evaluating code mutation.

Benchmark	Description	Input
bitcount	bit count algorithm	1125000
blowfish	encryption	large
dijkstra	path calculation	large
FFT	Fast Fourier Transformation	8 32768
jpeg	image (de)compression	large
qsort	quick sort algorithm	large
sha	secure hash algorithm	large
susan	image recognition	large

We also provide hardware performance results and consider three different Intel Pentium 4 machines. These machines differ in terms of their clock frequency, microarchitecture, memory hierarchy, and implementation technology; see Table 2.5 for the most significant differences.

2.6 Evaluation

We now evaluate the proposed code mutation approaches: (i) MA-CFO (memory access and control flow operation slicing) aiming at preserving the memory access and control flow behavior in the mutant, (ii) CFO (control flow operation slicing) aiming at preserving the control

Table 2.4: The baseline processor model considered in our simulations for evaluating code mutation.

Parameter	Configuration
ROB	128 entries
load queue	48 entries
store queue	32 entries
issue queues	4 16-entry issue queues
processor width	4 wide fetch, decode, dispatch, issue, commit
latencies	load (2), mul(3), div(20)
L1 I-cache	16 KB 4-way set-associative, 1 cycle
L1 D-cache	16 KB 4-way set-associative, 1 cycle
L2 cache	unified, 128 KB 16-way set-associative, 6 cycles
L3 cache	unified, 1 MB 16-way set-associative, 20 cycles
main memory	250 cycle access time
branch predictor	hybrid bimodal/gshare predictor
frontend pipeline	8 stages

Table 2.5: The Intel Pentium 4 machines used for evaluating code mutation.

		Memory	
Machine	Generation	L2 (KB)	Main (GB)
Pentium 4 2 GHz	Northwood	512	1
Pentium 4 2.8 GHz	Northwood	512	2
Pentium 4 3 GHz	Prescott	1024	1

flow behavior in the mutant only, and (iii) CFO plus ECF (CFO plus enforced control flow of infrequent branches). We evaluate the efficacy of these approaches along two dimensions: their ability to hide functional semantics, and their ability to preserve the performance characteristics in the mutant with respect to the proprietary application.

2.6.1 Hiding functional semantics

Static Instruction Ratio

Figure 2.7 shows results for the SIR metric, or the fraction of the application binary that can be mutated, which is an indication for how well binary mutation protects against static reverse engineering. There are four bars in this graph. The first bar quantifies the SIR metric by overwriting code that is not executed for the considered input set. On average, this results in a 44% SIR metric. The next three bars quantify the SIR metric for MA-CFO, CFO and CFO plus ECF code mutation; these approaches achieve a SIR metric of 56%, 60% and 62%, respectively. CFO achieves a higher SIR score than MA-CFO, and CFO plus ECF achieves a higher SIR score than CFO. The reason is that fewer slices need to be computed which increases the number of instructions eligible for code mutation.

The relative increase is limited though: we expected that the SIR metric would be much higher for CFO compared to MA-CFO. However, the relatively small increase seems to suggest that there is significant overlap between the slices for the memory accesses compared to the slices for the control flow operations. Not computing memory access slices does not reveal that many additional instructions eligible for code mutation. Put another way, by striving at preserving a program's control flow behavior, we also preserve most of the memory access behavior.

Another interesting note is that the achievable SIR is benchmark specific, and for some benchmarks more than 90% of the application binary can be mutated, see for example `gap` and `susan`. The high SIR score for `susan` and `qsort` correlates well with the small number of branches subject to slicing, see Figure 2.6. For other benchmarks though, the small number of branches subject to slicing does not translate into a high SIR score, see for example `bzip2`: a small number of control flow slices cover a large fraction of the entire program code.

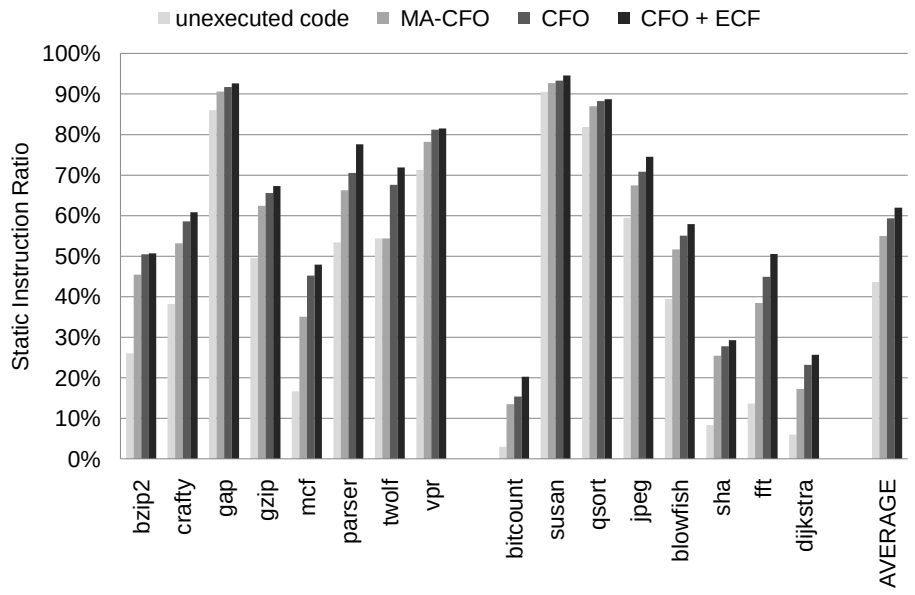

Figure 2.7: Quantifying the efficacy of benchmark mutation using the SIR (Static Instruction Ratio) metric. CFO plus ECF achieves an average SIR score of 62%.

Static Executed Instruction Ratio and Weighted Instruction Ratio

Figures 2.8 and 2.9 report similar results for the SEIR and WIR metrics, which are measures for how well code mutation protects against dynamic reverse engineering. The SEIR and WIR metrics have lower values than the SIR metric: average SEIR and WIR scores of 36% and 20%, respectively, compared to the average 62% SIR score for CFO plus ECF code mutation. Also, the WIR score is typically lower than the SEIR score. This suggests that code mutation primarily mutates code in less frequently executed code regions. The `susan` benchmark is an extreme example which has an SIR metric of 95%, an SEIR metric of 43% and a WIR metric of 8%. For other benchmarks on the other hand, such as `qsort`, code mutation mutates frequently executed code, and achieves a WIR score (53%) that is higher than its SEIR score (38%).

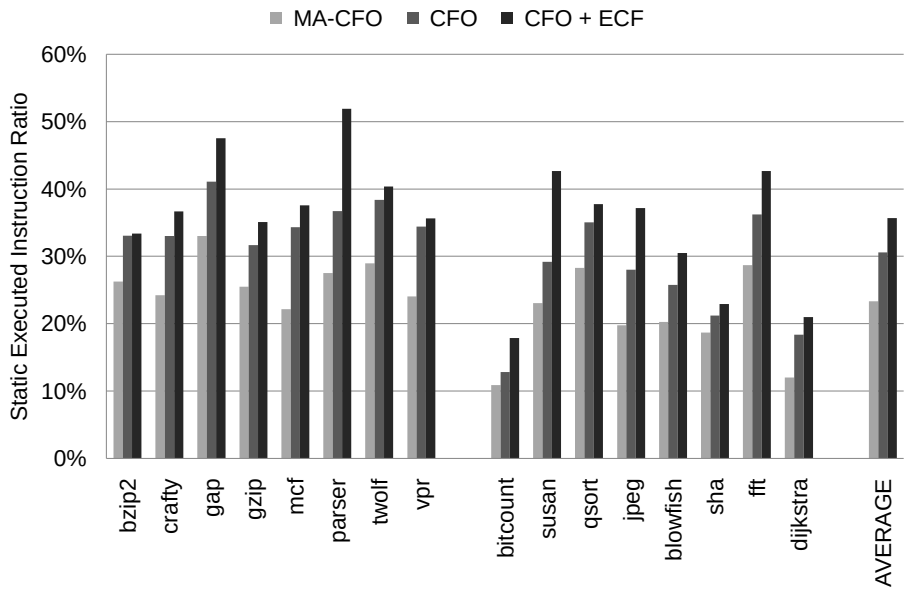

Figure 2.8: Quantifying the efficacy of benchmark mutation using the SEIR (Static Executed Instruction Ratio) metric. CFO plus ECF achieves an average SEIR score of 36%.

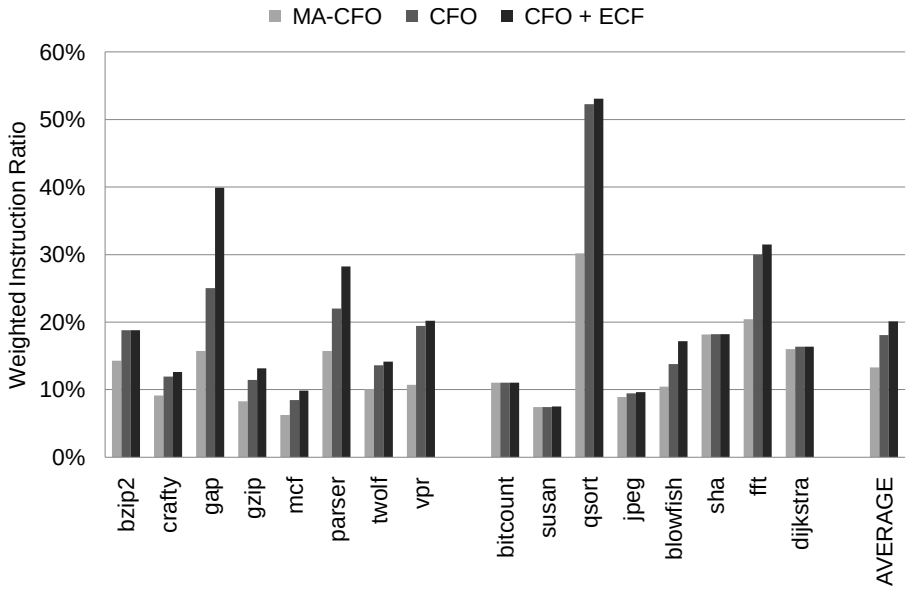

Figure 2.9: Quantifying the efficacy of benchmark mutation using the WIR (Weighted Instruction Ratio) metric. CFO plus ECF achieves an average WIR score of 20%.

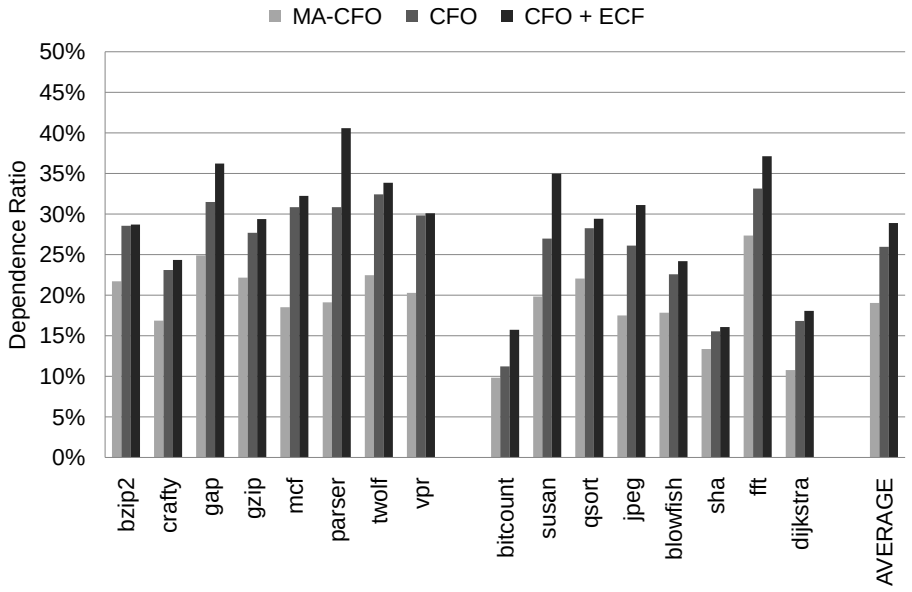

Figure 2.10: Quantifying the efficacy of benchmark mutation using the DR (Dependence Ratio) metric. CFO plus ECF achieves an average DR score of 29%.

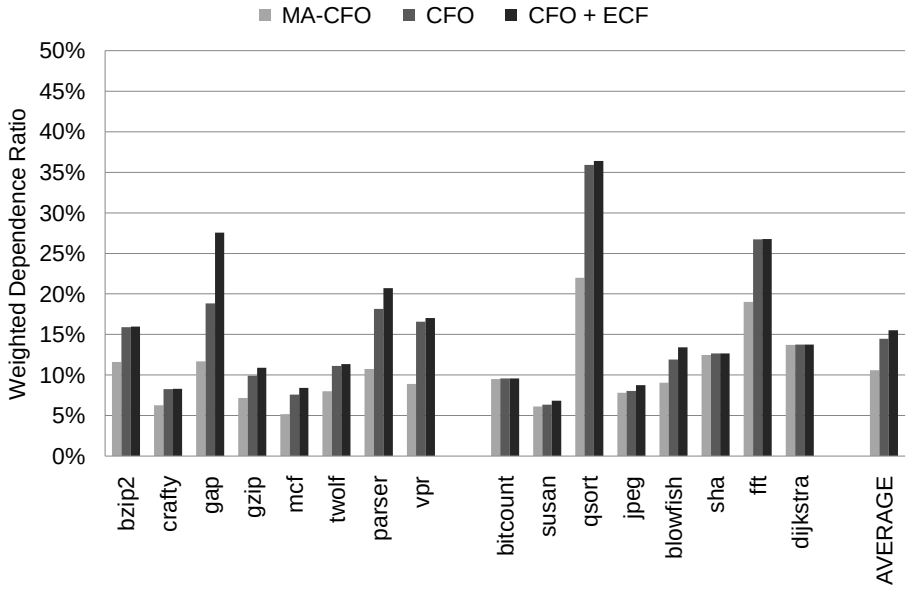

Figure 2.11: Quantifying the efficacy of benchmark mutation using the WDR (Weighted Dependence Ratio) metric. CFO plus ECF achieves an average WDR score of 16%.

Dependence Ratio and Weighted Dependence Ratio

Figures 2.10 and 2.11 show the DR and WDR metrics, respectively. The average DR and WDR metric scores are 29% and 16%, respectively, and go up to 40% and 35%, respectively. This result shows that a substantial fraction of the at run-time-exposed data dependencies are broken through code mutation, which will complicate reverse engineering significantly.

2.6.2 Performance characteristics

We now evaluate how well the mutant preserves the performance characteristics of the original application. We do this in three steps. We first consider our baseline simulated processor configuration, and subsequently evaluate how well the mutant tracks the original application across a microarchitecture design space. Finally, we consider three hardware platforms.

Simulation results

Figure 2.12 quantifies the execution time deviation for the mutant compared to the original application. The average (absolute) performance deviation equals 0.7%, 1.0% and 1.2% for MA-CFO, CFO, and CFO plus ECF, respectively. The maximum performance deviation is limited to 6%, see `qsort` which is also the benchmark with the highest WIR and WDR metric values.

We also evaluated code mutation across a microarchitecture design space in which we vary the cache hierarchy, see Table 2.6 for the different cache configurations. The average deviation across this design space equals 0.8%, 1.3% and 1.4% for MA-CFO, CFO and CFO plus ECF, respectively. The relative small increase in performance deviation suggests that CFO plus ECF is the mutation approach that represents a good trade-off in its ability to hide proprietary information while preserving performance characteristics.

This is further illustrated in Figure 2.13 and Figure 2.14. Figure 2.13 shows the normalized execution time for the original application across four cache hierarchy configurations for a four-wide superscalar pro-

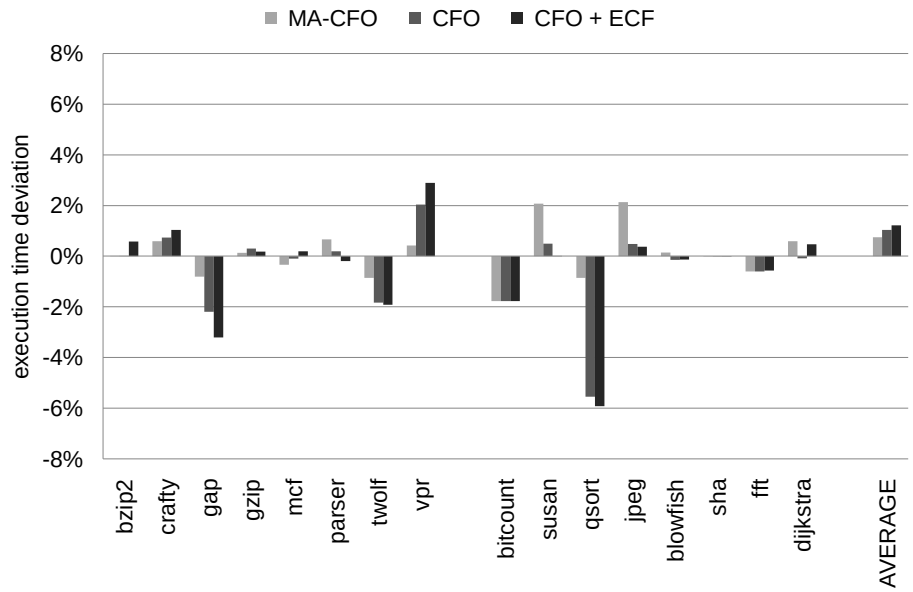

Figure 2.12: Execution time deviation for the mutant against the original application for the baseline processor configuration.

cessor, and Figure 2.13 shows the normalized execution time for the mutant considering CFO plus ECF. The mutant tracks the performance sensitivities across the memory hierarchy very well. For example, `mcf` benefits the most going from a 8 KB L1 cache to a 16 KB L1 cache, see Figure 2.13. This goes for the mutated version of `mcf` as well, see Figure 2.14

Table 2.6: Cache configurations used for evaluating code mutation.

	Cache size (KB)		
Configuration	L1	L2	L3
'small' cache	8	64	512
'medium-small' cache	16	128	1024
'medium-large' cache	32	256	2048
'large' cache	64	512	4096

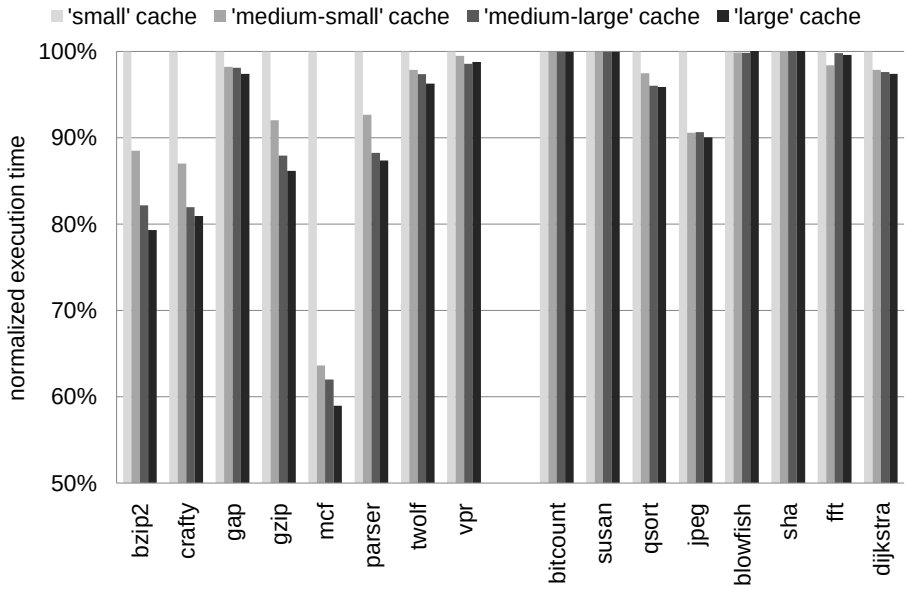

Figure 2.13: Normalized execution time for the original application across four cache hierarchy configurations for a four-wide superscalar processor.

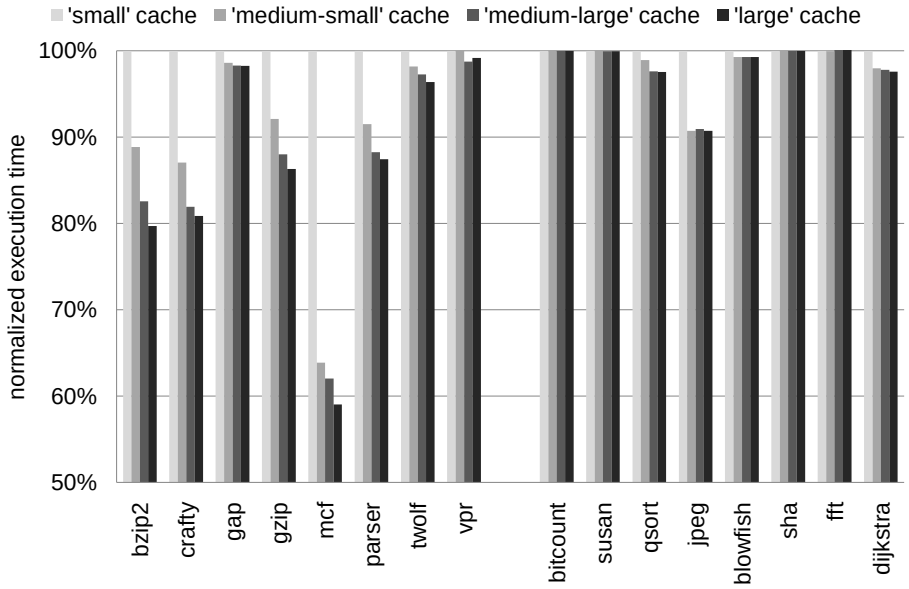

Figure 2.14: Normalized execution time for the mutant assuming CFO plus ECF across four cache hierarchy configurations for a four-wide superscalar processor.

Hardware results

The results presented so far are obtained through simulation. Figures 2.15 through 2.17 show results obtained from hardware measurements on three Intel Pentium 4 machines, see also Table 2.5.

Figure 2.15 shows the normalized execution times across the three Intel Pentium 4 machines for the original applications, and Figure 2.16 for the mutants; the results are normalized to the execution time of the original application on the 3.0 GHz Prescott machine. These results show that the mutants track the relative performance differences of the original application very well across the different hardware platforms. For example, for the SPEC benchmarks, `bzip2` and `mcf` show the largest performance improvement going from the 2.8 GHz Northwood machine to the 3.0 GHz Prescott machine; for the MiBench benchmarks, `susan` and `jpeg` seem to benefit the most from the Prescott architecture. We observe this for both the original applications (shown in Figure 2.15) and the benchmark mutants (shown in Figure 2.16).

Figure 2.17 quantifies the performance deviation of the mutant with respect to the original application on the Prescott Intel Pentium 4 (machine 3). The execution time deviation is small: 1.4% on average; the maximum deviation 6% is observed for `qsort`, which is the benchmark with the highest number of dynamically executed instructions that are mutated.

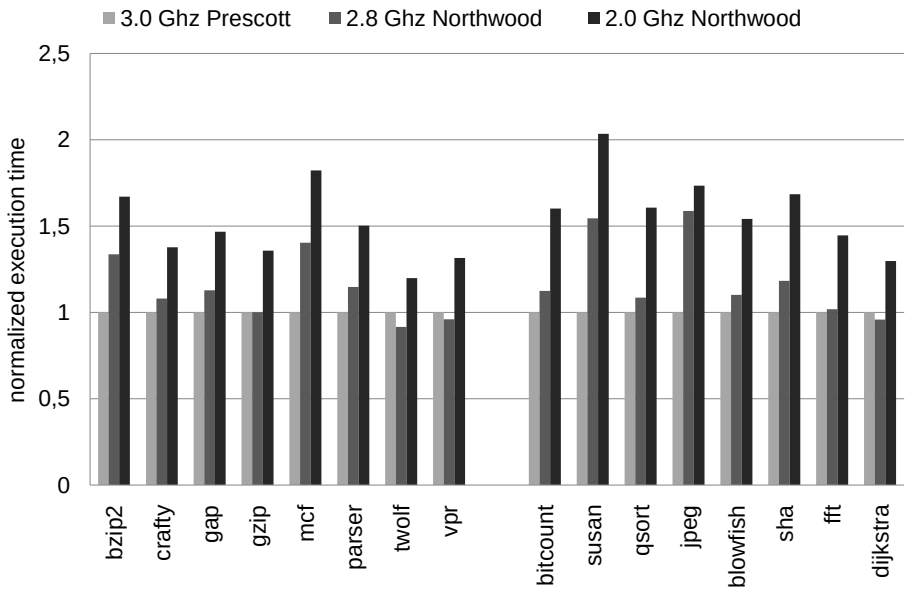

Figure 2.15: Normalized execution time for three hardware platforms for the original application; the results are normalized to the execution time of the original application on the 3.0 Ghz Prescott Intel Pentium 4 machine.

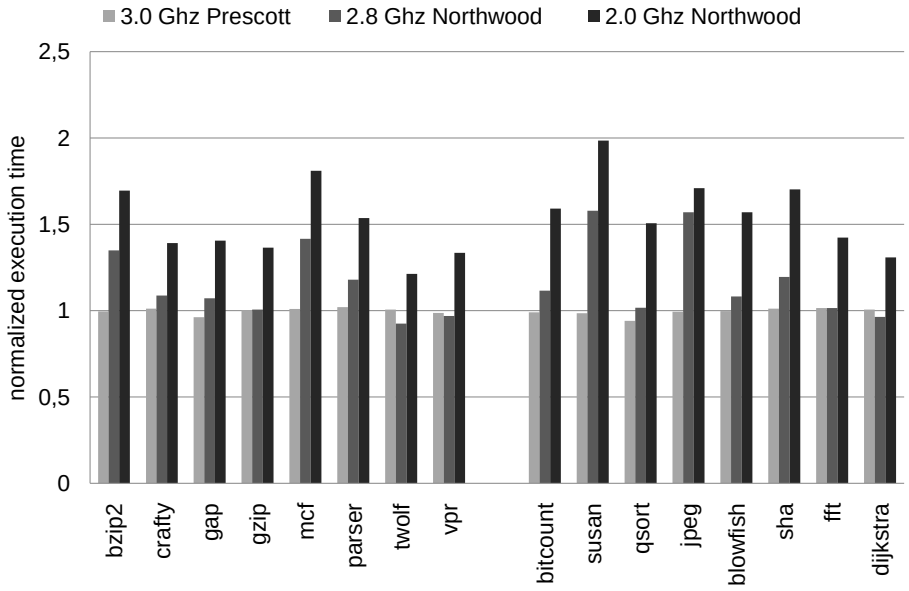

Figure 2.16: Normalized execution time for three hardware platforms for the mutant assuming CFO plus ECF; the results are normalized to the execution time of the original application on the 3.0 Ghz Prescott Intel Pentium 4 machine.

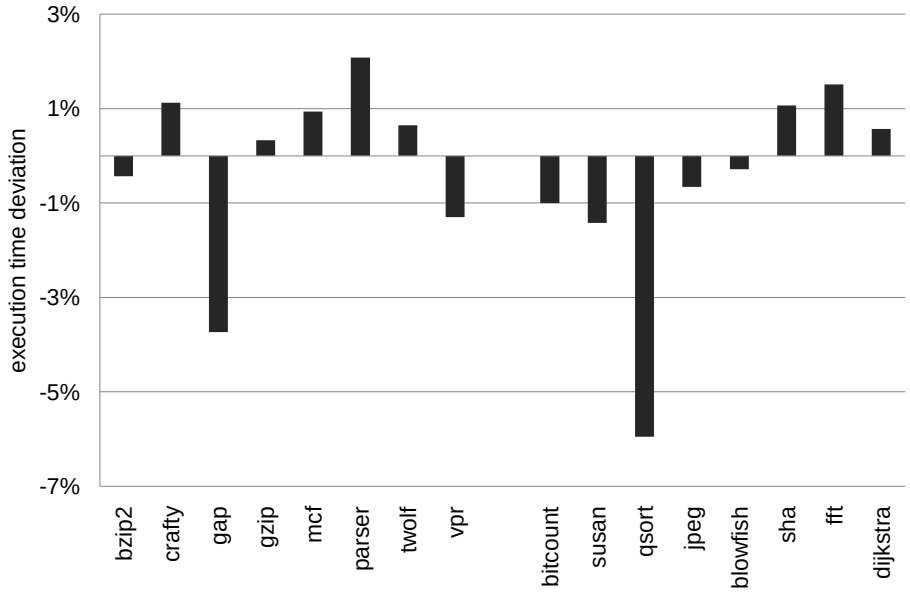

Figure 2.17: Execution time deviation for a mutant (CFO plus ECF) against its original application for the 3.0 Ghz Prescott Intel Pentium 4 machine.

2.7 Summary

Evaluating the performance of computer systems is typically performed with standardized, open-source benchmarks because industry vendors are reluctant to share their proprietary applications; however, open-source benchmarks may not be representative of these proprietary applications. This chapter presented code mutation, a novel methodology that enables automatic profile-based generation of benchmark mutants from binaries of proprietary applications in a way that hides functional semantics while preserving key performance characteristics. As such, these benchmark mutants can be distributed to third parties without exposing intellectual property, and then serve as proxies during benchmarking experiments.

Code mutation will be most useful for companies that offer in-house built services to remote customers. Such companies are reluctant to distribute their proprietary software. As an alternative, they could use mutated benchmarks as proxies for their proprietary software. The mutated benchmarks can help drive performance evaluation by third parties as well as guide purchasing decisions of hardware infrastructure. Being able to generate representative benchmark mutants without revealing proprietary information can also be an encouragement for industry to collaborate more closely with academia, i.e., it would make performance evaluation in academia more realistic and therefore more relevant for industry. In addition, developing benchmarks is both hard and time-consuming to do in academia, for which code mutation may be a solution.

We discussed and quantatively evaluated three code mutation approaches; these approaches differ in how well they preserve the proprietary application's memory access and control flow behavior in the mutant. We found CFO plus ECF to be the approach that represents the best trade-off between accuracy and information hiding. This approach computes control flow slices for frequently executed, non-constant branches, and mutates instructions that are not part of any of these slices. The slices are trimmed using constant value profiles to make more instructions eligible for code mutation. Our results obtained for a selection of SPEC CPU2000 and MiBench benchmarks report that up to 90% of the binary can be mutated, up to 50% of the dynamically executed instructions, and up to 35% of the at-run-time-exposed inter-operation data dependencies. We also demonstrated that the performance characteristics of the mutants correspond well

with those of the original applications; for CFO plus ECF, the average execution time deviation on hardware is 1.4%.

Chapter 3

Sampled simulation

Nothing is particularly hard if you divide it into small jobs.
Henry Ford

Code mutation conceals the intellectual property of an application, but it does not lend itself to the creation of short-running benchmarks. Sampled simulation on the other hand aims at reducing the simulation time of an application, and probably is the most widely used simulation acceleration technique today. After briefly revisiting sampled processor simulation, we propose NSL-BLRL, a technique that builds on No-State-Loss (NSL) and Boundary Line Reuse Latency (BLRL) for minimizing the cost associated with warming processor cache hierarchy state in sampled simulation. We extensively evaluate the accuracy of this technique and demonstrate that substantial simulation time speedups are obtained compared to prior work.

3.1 Sampled processor simulation

Computer architects and engineers rely heavily on detailed cycle-accurate simulation to explore and validate microarchitectural innovations. However, cycle-accurate simulators are extremely slow given the huge complexity of the microarchitectures that they model [Smith and Sohi, 1995]. Chiou et al. [2007] give an indication of the simulation speeds for a number of simulators that are widely used in computer architecture research and development, see Table 3.1. The reported speeds range from 1 KHz to a maximum of 740 Kilo Instructions per Second (KIPS), observed for `sim-outorder` — measured on an AMD Opteron 275 processor clocked at 2.2 GHz — which is part of the Sim-

Table 3.1: An overview of the simulation speeds for today's simulators [Chiou et al., 2007].

Simulator	Speed
Intel	1-10 KHz
AMD	1-10 KHz
IBM	200 KIPS
Freescale	80 KIPS
PTLSim [Yourst, 2007]	270 KIPS
Sim-outorder [Austin et al., 2002]	740 KIPS
GEMS [Martin et al., 2005]	69 KIPS

pleScalar ToolSet [Austin et al., 2002]. In other words, one second of native execution time corresponds to multiple hours (or even days) of simulation time.

At the same time, benchmarks that are being simulated grow in complexity as well, i.e., the dynamic instruction count of contemporary benchmarks increases in order to stress the increasingly more powerful processors. For the SPEC CPU benchmarks, the number of dynamically executed instructions per benchmark increased exponentially, as illustrated in Table 3.2. For example, the SPEC CPU89 benchmark suite executes an average of 2.5 billion instructions per benchmark, while for the SPEC CPU2006 benchmark suite, an average of about 2,500 billion instructions per benchmark need to be executed. Hence, we easily end up with months or even years of simulation, e.g., it takes PTLSim more than 3 months[1] to run a single SPEC CPU2006 benchmark to completion. And this is to simulate just a single microarchitecture design point. Clearly, these long simulation times preclude a detailed exploration of the design space.

Past research advocates sampling for speeding up detailed simulation [Laha et al., 1988] [Conte et al., 1996] [Sherwood et al., 2002] [Wunderlich et al., 2003] [Ekman and Stenström, 2005] [Luo et al., 2005]. The key idea of sampled simulation is to simulate only a small *sample* from a complete benchmark execution, as shown in Figure 3.1. A sample consists of one or more *sampling units*. We refer to the *pre-sampling units* as the parts between consecutive sampling units. Sampled simulation

[1]Note that the precise amount of time depends on the particular machine/benchmark used in the simulation experiment.

Table 3.2: The different SPEC CPU generations and their average dynamic instruction count [Phansalkar et al., 2004] [Joshi et al., 2006a] [Phansalkar et al., 2007].

Benchmark suite	Average dynamic instruction count (billion)
SPEC CPU89	≅ 2.5
SPEC CPU92	≅ 15
SPEC CPU95	≅ 75
SPEC CPU2000	≅ 230
SPEC CPU2006	≅ 2,500

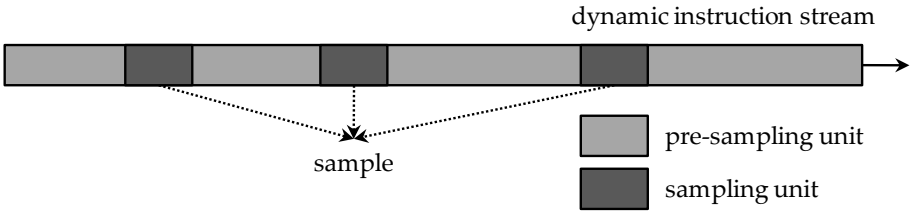

Figure 3.1: General concept of sampled simulation.

only simulates the instructions in the sampling units in a cycle-accurate manner instead of the entire dynamic instruction stream. Hence, significant speedups can be achieved.

There are three major challenges for sampled simulation to be accurate and fast:

1. **How to select the sampling units?** The first issue is the selection of representative sampling units. The challenge is to select sampling units in such a way that the sampled execution is an accurate picture of the complete execution of the program. In other words, sampling units should be chosen such that all major phases of the original program's execution are represented during sampled execution.

2. **How to initialize a sampling unit's architecture starting image?** The sampling unit's *Architecture Starting Image (ASI)* is the architecture state, i.e., register and memory state, needed to correctly functionally simulate the sampling unit's execution. This is not a

concern for trace-driven simulation[2] for which the instructions in the pre-sampling unit can be discarded from the trace. However, for execution-driven simulation[3], it is important to establish the correct architecture state at the beginning of the sampling unit as fast as possible so that the sampled simulation can quickly jump between sampling units.

3. **How to accurately estimate a sampling unit's microarchitecture starting image?** The sampling unit's *Microarchitecture Starting Image (MSI)* is the microarchitecture state (content of caches, branch predictor, etc.) at the beginning of the sampling unit. It is important to establish microarchitecture state at the beginning of each sampling unit in an efficient way. In literature, the unknown microarchitecture state is often referred to as the *cold-start* problem.

In this chapter, we address the cold-start problem by proposing a highly efficient and accurate technique for estimating the content of the caches (and cache-like structures such as translation lookaside buffers and branch targets buffers) at the beginning of a sampling unit. The main motivation to address this challenge is that a significant part of the total sampled simulation time is spent on establishing the microarchitecture state, as we will explain later on. As such, an efficient strategy to *warm* the microarchitecture state can yield significant simulation speedups. Before introducing our cache state warmup methodology, we briefly summarize proposed solutions to the first two challenges.

3.1.1 Selecting sampling units

There are two prevailing strategies for selecting sampling units: (i) statistical sampling and (ii) targeted sampling. Statistical sampling builds on statistical sampling theory to estimate the CPI error of the sampled simulation whereas targeted sampling builds on program analysis to select representative sampling units.

[2] In trace-driven simulation, a trace of program instructions and addresses is recorded and used to drive a software timing model of a microprocessor model. By consequence, functional simulation —which models the functional characteristics of an ISA — needs to be performed only once.

[3] An execution-driven simulator combines functional simulation with timing simulation to evaluate the performance of a microprocessor model — trace files do not need to be stored.

Statistical sampling

Statistical sampling selects the sampling units either randomly [Conte et al., 1996] or periodically. SMARTS (Sampling Microarchitecture Simulation) proposed by Wunderlich et al. [2003] is a well-known example of periodic sampling. They use the central limit theorem to determine how many sampling units are required to achieve a desired confidence interval at a given confidence level. The user first specifies an initial number of sampling units. After simulating these sampling units, the CPI error is estimated based on the central limit theorem. If the estimated CPI error is higher than the user-specified confidence interval, SMARTS then recommends a higher sampling frequency.

SMARTS uses a small sampling unit size of 1,000 instructions for SPEC CPU workloads, which implies that many sampling units are needed (typically in the order of 1,000 sampling units). This small sampling unit size also suggests that sampled simulation accuracy is very sensitive to the cold-start problem.

Targeted sampling

The idea of targeted sampling is to analyze a program's time-varying execution behavior and subsequently pick sampling units from each program phase. The SimPoint technique by Sherwood et al. [2002] is the most well-known targeted sampling approach. They propose to profile the program's execution in order to identify program phases, and select a sampling unit from each phase. Compared to SMARTS, a relative small number of sampling units are chosen (in the range of one up to ten sampling units), and the typical sampling unit size ranges from 1 M to 100 M instructions.

Yi et al. [2005] compare the SimPoint approach with the SMARTS approach. They conclude that SMARTS is slightly more accurate than SimPoint but SimPoint has a better speed versus accuracy trade-off.

3.1.2 Initializing the architecture state

There are two common approaches for establishing the architecture state (register and memory state) at the beginning of a sampling unit: (i) fast-forwarding and (ii) checkpointing. We first explain them since we will consider both scenarios for the evaluation of our cache warmup

strategy — once again, note that initializing the ASI does not apply to trace-driven simulation.

Fast-forwarding

The idea is to fast-forward between sampling units, i.e., navigate between sampling units through functional simulation. Functional simulation models only the functional characteristics of an instruction-set architecture, without updating the microarchitecture state and without computing performance metrics. In this scenario, sampled simulation begins fast-forwarding from the end of a sampling unit (or the beginning of the program) to construct the architecture starting image through functional simulation. When the beginning of the next sampling unit is reached, the simulator switches to detailed cycle-accurate simulation. At the end of the sampling unit, the simulator switches back to functional simulation (or quits in case the last sampling unit is simulated).

Obviously, functional simulation is much faster than detailed cycle-accurate simulation. Table 3.3 lists the simulation speeds for the SimpleScalar simulator models used in the evaluation [Austin et al., 2002]; the speeds are obtained on an AMD Athlon XP 2600+ processor. The `sim-fast` simulator model implements a functional simulator whereas `sim-outorder` is a detailed out-of-order performance simulator with a multi-level memory system. These results show that for the SimpleScalar ToolSet functional simulation is approximately ten times faster than cycle-accurate simulation — this is where the simulation speedup in sampled simulation (using fast-forwarding) comes from.

Checkpointing

The second scenario is checkpointing, which stores the architecture starting image before each sampling unit, i.e., the register contents and the memory state prior to a sampling unit. Simulating a sampling unit begins with loading its corresponding checkpoint from disk in order to update the register and memory state in the simulator. Subsequently, cycle-accurate simulation of the sampling unit begins.

Checkpointing avoids time-consuming fast-forwarding and allows for parallel simulation. However, the disadvantage is that full checkpoints can be very large, and thus costly in terms of disk space. For

Table 3.3: Simulation speeds for the SimpleScalar simulator models used in the evaluation: `sim-fast` (functional simulation), `sim-bpred` + `sim-cache` (functional warming), and `sim-outorder` (cycle-accurate simulation).

	SimpleScalar simulator model		
Benchmark	sim-fast (MIPS)	sim-bpred + sim-cache (MIPS)	sim-outorder (MIPS)
bzip2	7.78	5.63	0.57
crafty	6.57	5.08	0.68
eon	5.81	5.09	0.73
gcc	5.51	4.66	0.62
gzip	7.14	5.30	0.58
parser	6.71	5.40	0.65
twolf	7.53	4.98	0.87
vortex	6.10	4.97	0.72
vpr	6.30	5.22	0.80

example, Van Biesbrouck et al. [2005] report that full checkpoints can take up to 28.8 gigabytes for a single benchmark. They report an average file size per compressed checkpoint of 49.3 Mbytes. To combat these large checkpoints, they propose the *Touched Memory Image (TMI)*, which stores only the memory words that are read in a sampling unit. This yields a reduction in checkpoint size by more than two orders of magnitude. Similarly, Wenisch et al. [2006a] propose *live-points* to reduce the size of conventional checkpoints.

3.1.3 Initializing the microarchitecture state

The third issue in sampled simulation is to estimate the MSI (microarchitecture starting image) for the sampling unit to be simulated — to address the cold-start problem — as fast as possible while maintaining a sufficient level of accuracy. If the MSI for the sampling unit differs too much from the MSI that would have been obtained through detailed simulation of all the instructions prior to the sampling unit, the simulation results will be inaccurate. This is shown in Figure 3.2, where the IPC prediction error is presented when we assume an empty MSI

at the beginning of each sampling unit[4]. The IPC prediction error is computed as follows:

$$\mu_{error}(\%) = \frac{\left|IPC_{\textit{empty MSI}} - IPC_{\textit{perfect MSI}}\right|}{IPC_{\textit{perfect MSI}}}.$$

The error ranges from 17.5% (observed for `eon`) up to 47.2% (observed for `gzip`). Consequently, we must establish the MSI as accurate as possible to make sampled simulation feasible.

A naive approach for establishing the MSI is to functionally-warm the microarchitectural state for the entire duration between sampling units. Functional warming is a combination of functional simulation with specialized branch predictor and cache hierarchy simulation — simulation speed is slower than functional simulation but faster than detailed simulation, as shown in Table 3.3. The disadvantage of this approach though is that functional warming quickly becomes a performance bottleneck. This is because the pre-sampling unit size is typically much larger than the sampling unit size [Wunderlich et al., 2003], i.e., the faster simulation speed in the pre-sampling unit does not offset the larger number of instructions that needs to be simulated. In other words, reducing the time spent on establishing the MSI can decrease the total simulation time significantly. Therefore, it is important to study efficient but accurate warmup strategies.

3.2 Cache state warmup

The most critical aspect of the microarchitecture state are the caches and cache-like structures such as TLBs and BTBs. The reason is that cache structures can be large and thus can introduce a long history. Therefore, researchers have proposed several cache state warmup techniques for approximating the cache state at the beginning of a sampling unit. These techniques can be divided in two important categories: (i) adaptive warming and (ii) checkpointed warming. The cache state warmup strategy that we propose uses insights obtained from both approaches.

[4]We consider a sampling unit size of 1 M instructions and a pre-sampling unit size of 100 M instructions. More details about the experimental setup can be found in Section 3.4.

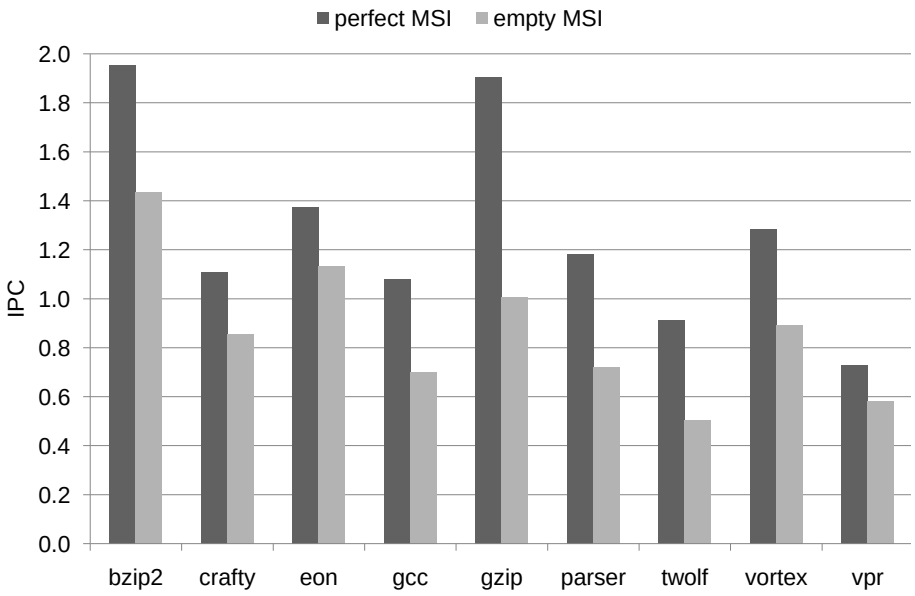

Figure 3.2: IPC prediction error considering an empty microarchitectural starting image (MSI).

3.2.1 Adaptive warming

Functionally-warming the microarchitectural state for the entire duration between consecutive sampling units is usually not necessary. The key idea of adaptive warming is to approximate the microarchitecture state with a reduced functional warming period. The challenge lies in determining the optimal length of this reduced functional warming period: underestimating the warming period leads to inaccurate simulation results while overestimating the warming period sacrifices simulation speed.

The best well-known technique for determining cache warming requirements is Memory Reference Reuse Latency (MRRL). MRRL also forms the base for Boundary Line Reuse Latency (BLRL) and ultimately NSL-BLRL. Therefore, we detail on these cache state warmup strategies; in addition, we give pointers to other warmup strategies as well.

Memory Reference Reuse Latency (MRRL)

Haskins and Skadron [2003] propose Memory Reference Reuse Latency (MRRL) for accurately warming up microarchitectural state at the be-

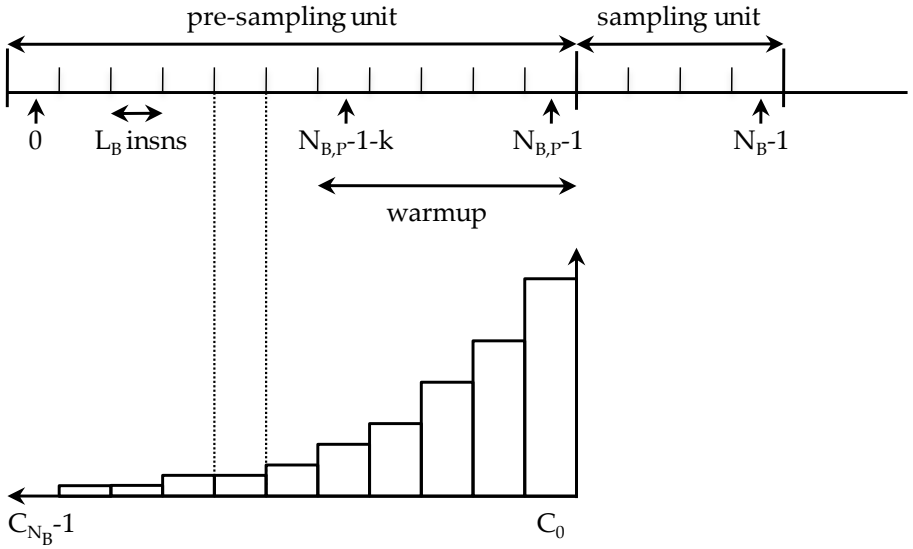

Figure 3.3: Determining warmup using MRRL (Memory Reference Reuse Latency).

ginning of each sampling unit. As suggested, MRRL refers to the number of instructions between consecutive references to the same memory location, i.e., the number of instructions between a reference to address 'A' and the next reference to 'A'. The MRRL warmup approach computes the MRRL for each memory reference in the sampling unit. Subsequently, these MRRLs are used to build a histogram. For this purpose, they divide the pre-sampling/sampling unit pair into N_B non-overlapping buckets each containing L_B contiguous instructions; in other words, the total pre-sampling unit/sampling unit pair consists of $N_B \cdot L_B$ instructions; see also Figure 3.3. The buckets receive an index from 0 to $N_B - 1$ in which index 0 is the first bucket in the pre-sampling unit. The first $N_{B,P}$ buckets constitute the pre-sampling unit and the remaining $N_{B,S}$ buckets constitute the sampling unit; obviously, $N_B = N_{B,P} + N_{B,S}$.

The MRRL warmup strategy also maintains N_B counters c_i ($0 \leq i < N_B$). These counters c_i will be used to build the histogram of MRRLs. Through profiling, the MRRL is calculated for each reference and the associated counter is updated accordingly. For example, for a bucket size $L_B = 10,000$ (as is the case in [Haskins and Skadron, 2003]) an MRRL of 124,534 will increment counter c_{12}. When the complete pre-sampling unit/sampling unit pair is profiled, the MRRL histogram p_i,

$(0 \leq i < N_B)$ is computed. This is done by dividing the bucket counters by the total number of references in the pre-sampling unit/sampling unit pair:

$$p_i = \frac{c_i}{\sum_{j=0}^{N_B-1} c_j}.$$

In other words, p_i represents the probability for observing an MRRL between $i \cdot L_B$ instructions and $(i+1) \cdot L_B - 1$ instructions:

$$p_i = Prob[i \cdot L_B < MRRL \leq (i+1) \cdot L_B - 1].$$

Not surprisingly, the largest $p_i's$ are observed for small values of i due to the notion of temporal locality in computer program address streams. Using the histogram p_i, Haskins and Skadron calculate the bucket corresponding to a given percentile $K\%$, i.e., bucket k for which $\sum_{m=0}^{k-1} p_m < K\%$ and $\sum_{m=0}^{k} p_m \geq K\%$. This means that of all the references in the current pre-sampling unit/sampling unit pair, $K\%$ have a reuse latency that is smaller than $k \cdot L_B$. Hence, Haskins and Skadron define these k buckets as their *warmup* buckets. In other words, functional warming is started $k \cdot L_B$ instructions before the sampling unit.

An important limitation in efficiency of MRRL is that a mismatch in the MRRL behavior in the pre-sampling unit versus the sampling unit might result in a suboptimal warmup strategy in which the warmup is either too short to be accurate, or too long for the attained level of accuracy. For example, if the reuse latencies are generally larger in the sampling unit than in the pre-sampling unit/sampling unit pair, the warmup will be too short and by consequence, the accuracy might be poor. On the other hand, if reuse latencies are generally shorter in the sampling unit than in the pre-sampling unit/sampling unit pair, the warmup will be too long for the attained level of accuracy. One way of solving this problem is to choose the percentile $K\%$ to be large enough. The result is that the warmup will be longer than needed for the attained accuracy.

Boundary Line Reuse Latency (BLRL)

Boundary Line Reuse Latency (BLRL) [Eeckhout et al., 2005] improves upon MRRL. In BLRL, the sample is scanned for reuse latencies that cross the pre-sampling unit/sampling unit boundary line, i.e., a memory location is referenced in the pre-sampling unit and the next reference to the same memory location is in the sampling unit. For each

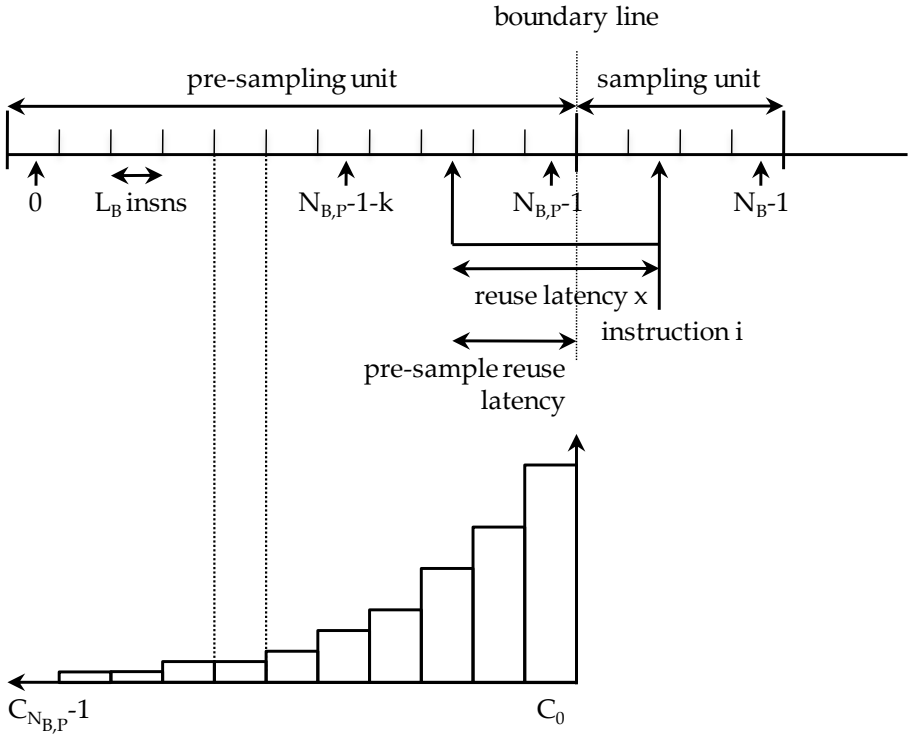

Figure 3.4: Determining warmup using BLRL (Boundary Line Reuse Latency).

of these cross BLRLs, the *pre-sampling unit reuse latency* is calculated. This is done by subtracting the distance in the sampling unit from the MRRL. For example, if instruction i has a cross BLRL x, the pre-sampling unit reuse latency then is $x - (i - N_{B,P} \cdot L_B)$; see Figure 3.4. A histogram is built up using these pre-sampling reuse latencies. As is the case for MRRL, BLRL uses $N_{B,P}$ buckets of size L_B to limit the size of the histogram. This histogram is then normalized to the number of reuse latencies crossing the pre-sampling unit/sampling unit boundary line. The required warmup length is then computed to include a given percentile $K\%$ of all reuse latencies that cross the pre-sampling unit/sampling unit boundary line.

There are three differences between BLRL and MRRL. First, BLRL considers reuse latencies for memory references originating from instructions in the sampling unit whereas MRRL considers reuse latencies for memory references originating from instructions both in the pre-sampling unit *and* the sampling unit. Second, BLRL only considers

reuse latencies that cross the pre-sampling unit/sampling unit boundary line; MRRL considers all reuse latencies. Third, in contrast to MRRL which uses the reuse latency to update the histogram, BLRL uses the pre-sampling reuse latency. Previous work [Eeckhout et al., 2005] has shown that BLRL substantially outperforms MRRL; the warmup length of BLRL is nearly half the warmup length of MRRL for the same level of accuracy.

Other adaptive warming approaches

Full warmup continuously keeps the cache state warm between sampling units. This is a very accurate approach but increases the time spent between sampling units. This approach is implemented in SMARTS [Wunderlich et al., 2003].

Luo et al. [2005] propose a self-monitored adaptive cache warmup scheme in which the simulator monitors the warm-up process of the caches and decides when the caches are warmed up based on simple heuristics. The limitation of this approach is that it is a priori unknown when the caches will be warmed up and when detailed simulation should get started, which is an issue for periodic sampling and targeted sampling.

3.2.2 Checkpointed warming

In checkpointed warming, the idea is to checkpoint or to store the microarchitecture state at the beginning of each sampling unit and impose this state during sampled simulation. This approach yields a perfect MSI. However, the storage needed to store these checkpoints can explode in case many sampling units are required. In addition, the microarchitecture state needs to be stored for each specific hardware configuration. For example, for each cache configuration a checkpoint needs to be made. Obviously, the latter constraint implies that the complete program execution needs to be simulated for these various hardware structures. Since this is infeasible to do in practice, researchers have proposed more efficient approaches to microarchitecture state checkpointing.

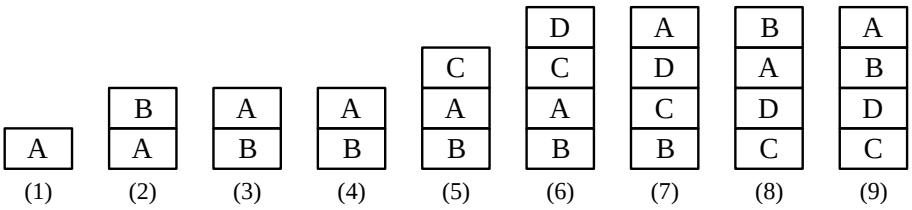

Figure 3.5: Building the LRU stack for 'ABAACDABA'.

No-State-Loss (NSL)

One example is the No-State-Loss (NSL) approach [Conte et al., 1998] which scans the pre-sampling unit and records the latest reference to each unique memory location in the pre-sampling unit. This is the stream of unique memory references as they occur in the memory reference stream sorted by their least recently usage. In fact, NSL keeps track of all the memory references in the pre-sampling unit and then retains the last occurrence of each unique memory reference. The resulting stream is referred to as the *Least Recently Used (LRU) stream*.

Consider the example reference stream 'ABAACDABA'; the alphabetic characters are assumed to represent memory addresses. The LRU stream of this reference stream is 'CDBA'. Computing this LRU stream can be done by building the LRU stack for the given reference stream, as illustrated in Figure 3.5. The LRU stack operates as follows: if address 'X' from the reference stream is not present on the stack, it is pushed onto the stack. For example, 'A' is pushed onto the stack in the first step, followed by 'B' in the second step, etc. If, in contrast, address 'X' is already present on the stack, it is removed from the stack and repushed onto the stack. See for example the third step, where 'A' moved from the bottom of the stack to the top of the stack. In the fourth step, 'A' was already on the top of the stack. Continuing in this manner results in the LRU stream 'CDBA', as illustrated in Figure 3.5 (9).

Both the original reference stream and the LRU stream yield the same state when applied to a cache with an LRU replacement policy. The No-State-Loss warmup method exploits this property by computing the LRU stream of the pre-sampling unit and by applying this stream to the cache as warmup. By consequence, the No-State-Loss warmup strategy yields perfect warmup for caches with an LRU replacement policy.

Other checkpointed warming approaches

Barr et al. [2005] extend the NSL approach for reconstructing the cache and directory state during sampled multiprocessor simulation. In order to do so, they keep track of a timestamp per unique memory location that is referenced. In addition, they keep track of whether accessing the memory location originates from a load or a store operation. This information allows them to quickly reconstruct the cache and directory state at the beginning of a sampling unit [Barr et al., 2005].

Van Biesbrouck et al. [2005] propose the *Memory Hierarchy State (MHS)* approach, which simulates the largest cache of interest once for the entire program execution, and stores a checkpoint of the cache content at the start of each sampling unit. The checkpoint is stored in a manner that allows them to faithfully recreate the content of smaller caches, i.e., caches with smaller sized memory hierarchies (smaller associativity/reduced number of sets). Hence, the MHS needs to be collected only once for each block size and replacement policy. A similar checkpointing approach is proposed by Wenisch et al. [2006a], referred to as *live-points*.

3.3 Combining NSL and BLRL into NSL-BLRL

We propose a hybrid cache state warmup approach that combines MSI checkpointing through NSL with BLRL into NSL-BLRL [Van Ertvelde et al., 2006] [Van Ertvelde et al., 2008]. This is done by computing both the LRU stream as well as the BLRL warmup buckets corresponding to a given percentile $K\%$. Only the unique references (identified through NSL) that are within the warmup buckets (determined through BLRL) will be used to warmup the caches. This could be viewed as pruning the LRU stream with BLRL information, as illustrated in Figure 3.6. The LRU stream of the memory reference stream in the pre-sampling unit is 'CDBA' (as computed previously). Also, BLRL recommends to begin functional warming at memory reference 'D' — we assume a K value of 75%. Hence, 75% of all reuse latencies that cross the boundary line between the pre-sampling unit and the sampling unit are included in the warmup period — memory reference 'C' is not part of this warmup period. This information is then used to construct the reduced NSL-BLRL checkpoint 'DBA', which no longer contains memory reference 'C', as shown in Figure 3.6(c).

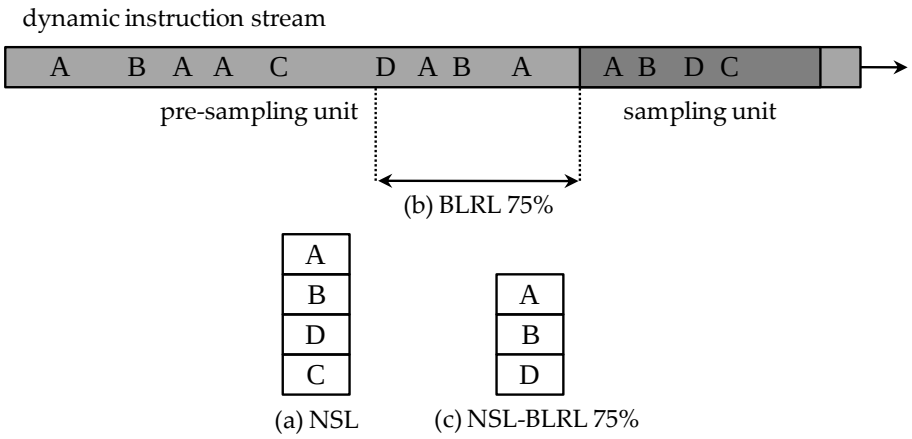

Figure 3.6: Combining NSL and BLRL into NSL-BLRL.

NSL-BLRL could also be viewed of as computing the LRU stream from the BLRL warmup buckets. Consider once again the example shown in Figure 3.6: BLRL 75% prescribes to begin functional warming at memory reference D, as mentioned above. Consequently, memory reference stream 'DABA' is used to warmup the caches. The LRU stream of this reference stream is 'DBA' — this forms the same NSL-BLRL checkpoint as calculated previously.

Integrating NSL and BLRL can be done without significantly increasing the complexity of the warmup procedure. Computing the LRU stream requires building and maintaining an LRU stack. Searching the LRU stack for the last reference to a given memory location can be done efficiently using a hash table; the hash table uses the memory address as its index and returns a pointer to the LRU stack entry. The same hash table can also be used to simultaneously identify the last reference in the dynamic instruction stream to the same memory location — next to returning a pointer to the LRU stack, the hash table then as well returns the position in the dynamic instruction stream. The location of the last reference in the dynamic instruction stream compared to the current memory access then determines the BLRL distance. Figure 3.7 illustrates this with an example. The input to the hash function is memory address 'A' (used by instruction 13); the hash of this memory address (a) is used to locate the last reference to 'A' in the LRU stack, i.e., the value at memory address a points to the bottom of the LRU stack (0x0). From the same hash table, it also appears that instruction 9 was the last instruction that referred to memory address 'A'. As such, the BLRL

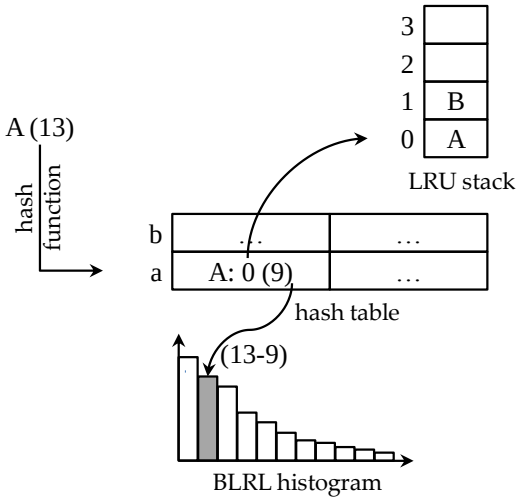

Figure 3.7: Integrating NSL and BLRL using a single hash table.

distance equals 4 (13-9) — this information is then used to update the BLRL histogram. In summary, NSL and BLRL can be implemented efficiently using a single hash table, which needs to be performed only once.

Using NSL-BLRL as a warmup approach, the subsequent operation is as follows. The reduced LRU stream as it is obtained through NSL-BLRL is to be stored on disk as a MSI checkpoint. Upon simulating a sampling unit, the reduced LRU stream is loaded from disk, the cache state is warmed up and finally, the simulation of the sampling unit begins. Note that NSL-BLRL can be used with both fast-forwarding and checkpointing.

The advantage of NSL-BLRL over NSL is that NSL-BLRL requires less disk space to store the warmup memory references; in addition, the smaller size of the reduced LRU stream results in faster warmup processing. The advantage over BLRL is that loading the reduced LRU stream from disk is more efficient than functional warming. According to our results, the warmup length for BLRL is at least two orders of magnitude longer than for NSL-BLRL. As such, significant speedups are obtained compared to BLRL. Compared to existing checkpointing techniques, NSL-BLRL is more broadly applicable during design space exploration. Both the MHS approach [Van Biesbrouck et al., 2005] and the live-points approach [Wenisch et al., 2006a] require the cache line

size to be fixed, i.e., if a cache needs to be simulated with a different cache line size, the warmup info needs to be recomputed. On the other hand, NSL-BLRL inherits the limitation from NSL of only guaranteeing perfect warmup for caches with LRU replacement. Caches with other replacement policies such as random, first-in first-out (FIFO), not-most-recently-used (NMRU) are not guaranteed to get a perfectly warmed up cache state under NSL-BLRL (as is the case for NSL) — however, the difference in warmed up hardware state is very small, as we show experimentally in the evaluation section.

3.4 Experimental setup

For the evaluation we use 9 SPEC CPU2000 integer benchmarks, see Table 3.4. The binaries, which were compiled and optimized for the Alpha 21264 processor, were taken from the SimpleScalar website[5]. All measurements presented in the evaluation section are obtained using the MRRL software[6] which in its turn is based on the SimpleScalar software [Austin et al., 2002]. The processor simulation model is shown in Table 3.5. The caches use write-allocate and write-back policies. We consider 50 sampling units each containing 1 M instructions. We select a sampling unit every 100 M instructions unless mentioned otherwise. These sampling units were taken from the beginning of the program execution to limit the simulation time while evaluating the various warmup strategies with varying percentiles $K\%$. Taking sampling units deeper down the program execution would have been too time-consuming given the large fast-forwarding needed. However, we believe this does not affect the conclusions, since the warmup strategies that are evaluated can be applied to any collection of sampling units. Once a set of sampling units is provided, either warmup strategy can be applied to it.

We quantify the accuracy of a warmup strategy using the IPC prediction error, i.e., the relative difference between the IPC for perfect warmup against the IPC for the warmup strategy of interest. The warmup length is defined as the number of instructions under functional warming, i.e.m functional simulation while updating the MSI.

[5] `http://www.simplescalar.com`
[6] `http://www.cs.virginia.edu/~jwh6q/mrrl-web`

Table 3.4: The SPEC CPU2000 integer benchmarks used along with their input.

Benchmark	Description	Input
bzip2	compression	program
crafty	game playing: chess	ref
eon	computer visualization	rushmeier
gcc	C programming language compiler	integrate
gzip	compression	graphic
mcf	combinatorial optimization	lgred
parser	word processing	ref
twolf	place and route simulator	ref
vpr	FPGA circuit placement and routing	route

Table 3.5: The baseline processor simulation model.

Parameter	Configuration
instruction cache	16 KB, 2-way set associative, 32-byte block, 2 cycles access latency
data cache	32 KB, 4-way set associative, 32-byte block, 2 cycles access latency
unified L2 caches	1 MB, 4-way set associative, 32-byte block, 20 cycles access latency
I-TLB and D-TLB	32-entry 8-way set-associative with 4 KB pages
memory	150 cycle round trip access
branch predictor	8 K-entry hybrid predictor selecting between an 8K-entry bimodal predictor and a 2-level (8 K $\times$ 8 K) local branch predictor
speculative update	at dispatch time
branch misprediction penalty	14 cycles
IFQ	32-entry instruction fetch queue
RUU and LSQ	128 entries and 32 entries, respectively
processor width	8 issue width, 8 decode width, 8 commit width
functional units	8 integer ALUs, 4 load/store units, 2 fp adders, 2 integer and 2 fp mult/div units

3.5 Evaluation

In this section, we extensively evaluate our NSL-BLRL approach and compare it with NSL and BLRL. We have a number of criteria to evaluate our improved warmup proposal, namely: accuracy, number of warm simulation instructions, overall simulation time and storage requirements.

3.5.1 Accuracy

Our first criterion to evaluate NSL-BLRL is its accuracy. Figure 3.8 shows the IPC prediction error for BLRL and NSL-BLRL for the considered benchmarks and for varying percentiles $K\%$ (note that NSL yields the same accuracy as NSL-BLRL 100%). The IPC prediction error is the relative error compared to continuous warmup, i.e., all instructions prior to the sampling unit are functionally warmed. As reported in previous work [Eeckhout et al., 2005], BLRL results in highly accurate warmup. BLRL yields small IPC prediction errors of only a few percent. Especially for large percentiles $K\%$, the IPC prediction error because of an inaccurate MSI is very small. For example, for BLRL 95%, the maximum error is only 1.6% (`twolf`). For BLRL 100%, the error is almost zero. Comparing NSL-BLRL 100% versus BLRL 100% typically gives slightly higher IPC prediction errors; however, the difference is very small, at less than 1%. There are two reasons for these slightly higher IPC prediction errors. The first reason is that while warming the caches through NSL-BLRL we do not keep track of dirty cache blocks, whereas BLRL does keep track of dirty cache blocks. Our results show that not warming dirty cache block info has a small impact on overall accuracy. This is to be expected given the fact that contemporary out-of-order microprocessors give priority to load operations over writing back dirty data to higher level caches in the memory hierarchy. If needed, warming dirty cache blocks can be supported by storing status information for each cache block so that dirty cache blocks are correctly marked, and dirty cache misses are modeled as such.

The second reason for the difference between the NSL-BLRL and BLRL is that NSL-BLRL only warms the cache state but does not warm branch predictor state. BLRL on the other hand warms both the cache hierarchy and branch predictor state. However, for the considered sampling unit size, we found this influence to be small. To experimentally verify this, we compared the accuracy of NSL-BLRL versus BLRL for

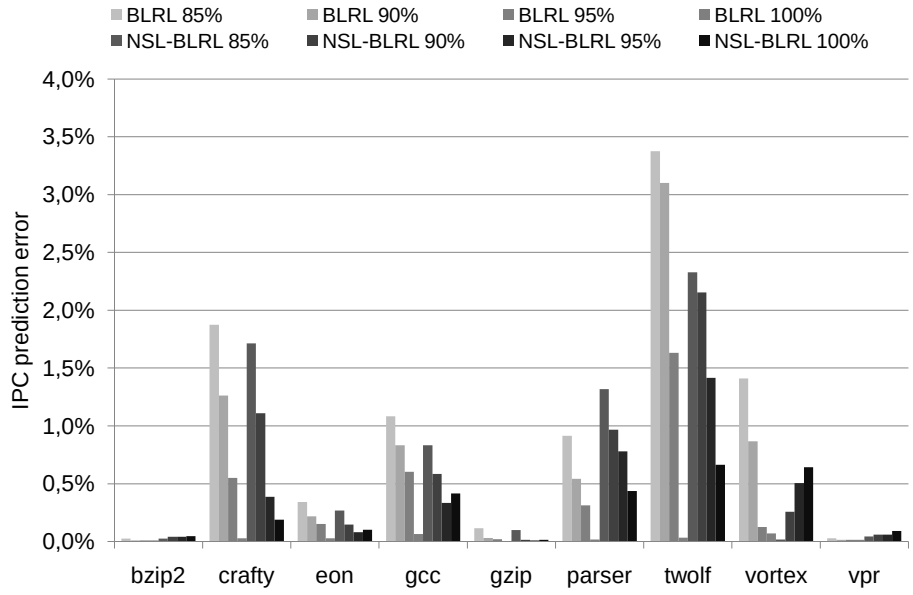

Figure 3.8: IPC prediction error for BLRL versus NSL-BLRL.

perfect branch predictors — this was to exclude the branch predictor component in the warmup state — and we obtained very similar results to what is being reported in Figure 3.8. Hence, we conclude that the impact of the branch predictor state is small.

In summary, we conclude that NSL-BLRL is a highly accurate cache warmup approach that is nearly as accurate as BLRL. Especially, high percentiles $K\%$ yield highly accurate performance estimates. The maximum error for $K = 95\%$ equals 1.4% (`twolf`); for $K = 100\%$, the maximum error is even less, 0.66% (also for `twolf`).

3.5.2 Warmup length

Figure 3.9 shows the number of warm simulation instructions for BLRL as well as the number of warm simulation references for NSL and NSL-BLRL. For BLRL and NSL-BLRL, we consider different percentiles $K\%$. Note that the vertical axis is on a logarithmic scale. We observe that NSL-BLRL yields a reduction in the number of warm simulation instructions by two to three orders of magnitude compared to BLRL. The reason for this dramatic reduction is that the number of warm simulation instructions for NSL-BLRL is proportional to the number of unique

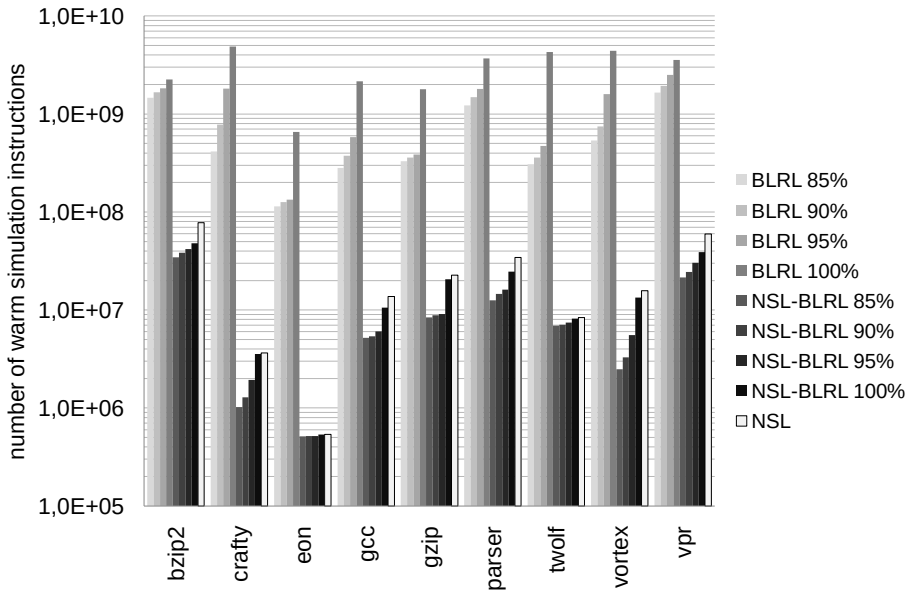

Figure 3.9: The number of warm simulation instructions for BLRL versus the number of warm simulation references for NSL-BLRL and NSL.

references in the pre-sample. BLRL on the other hand uses all references from a given warmup starting point up to the sampling unit starting point. Note that these results were obtained for 100 M instruction pre-samples prior to each sampling unit. For larger pre-sampling units, the difference in the number of warm simulation instructions is likely to increase when comparing BLRL versus NSL-BLRL.

Comparing NSL-BLRL versus NSL we also observe a substantial decrease in the number of warm simulation instructions. Figure 3.10 shows the number of warm simulation references for NSL-BLRL as a fraction of NSL. Some benchmarks do not benefit substantially from NSL-BLRL compared to NSL. However, we observe that NSL-BLRL 100% yields substantial warm simulation reductions for other benchmarks — up to 39% for `bzip2`; i.e., the warmup length for NSL-BLRL 100% is 61% of the NSL warmup length. For smaller $K\%$ percentiles, the reduction in warmup length increases significantly.

The results are given for a pre-sampling unit size of 100 M instructions. For larger pre-sampling unit sizes, the benefit for NSL-BLRL over NSL in terms of the number of warm simulation instructions even increases. This is illustrated in Figure 3.11 where the number of warm

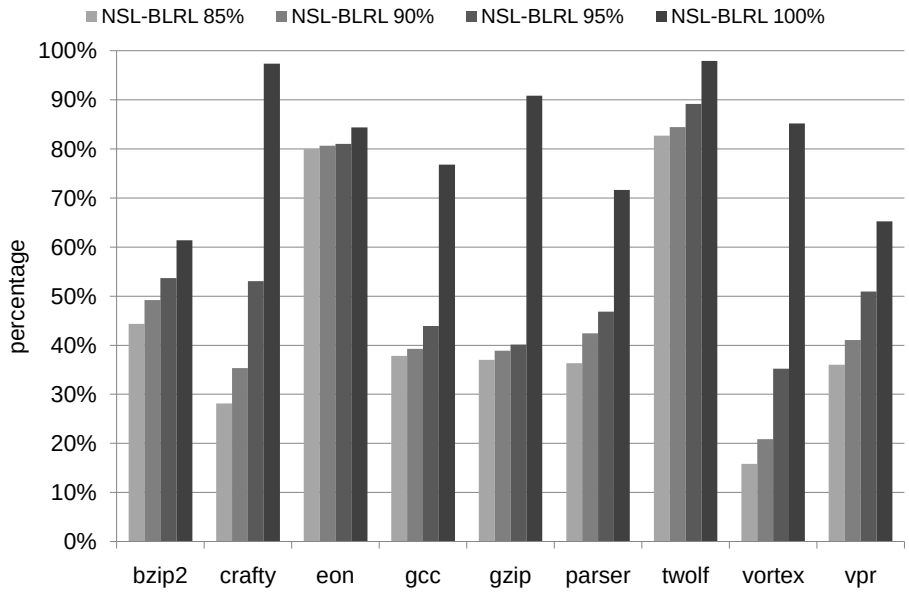

Figure 3.10: The number of warm simulation references for NSL-BLRL as a fraction of the number of warm simulation references for NSL.

simulation instructions is shown as a function of the pre-sampling unit sizes for NSL and NSL-BLRL for `bzip2` — similar curves were obtained for other benchmarks. The important trend to be observed from this graph is that the number of warm simulation instructions does not increase as fast for NSL-BLRL as it does for NSL. As such, we can conclude that NSL-BLRL is more scalable for larger pre-sampling unit sizes and thus, longer running applications.

3.5.3 Simulation time

The number of warm simulation instructions only gives a rough idea about the impact of the warmup strategies on overall simulation time. To evaluate the simulation time speedup, we consider two scenarios[7] for sampled simulation : (i) establishing the ASI (Architectural Starting Image) through fast-forwarding and (ii) establishing the ASI through checkpointing.

[7] As already explained in Subsection 3.1.2.

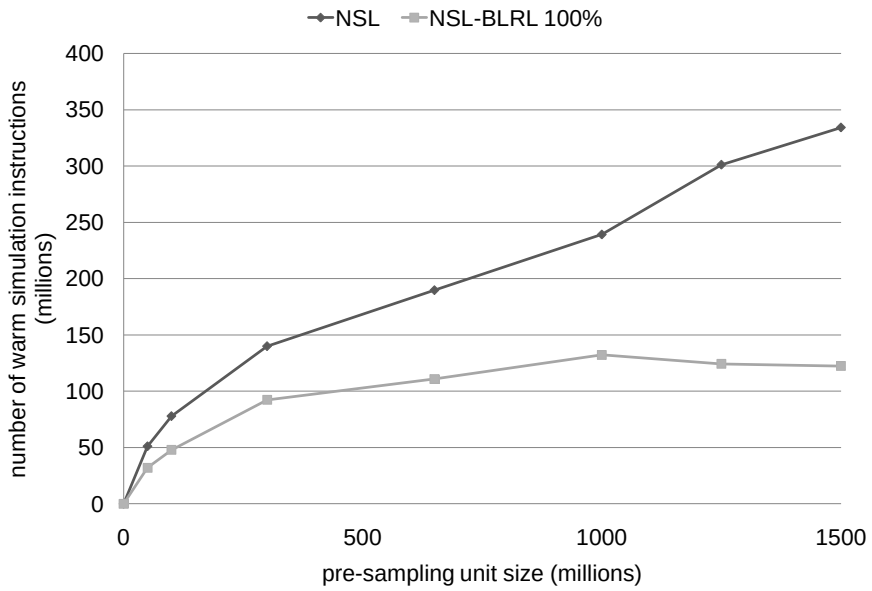

Figure 3.11: The number of warm simulation references for NSL-BLRL 100% and NSL as a function of the pre-sampling unit size for `bzip2`.

Fast-forwarding

The results in Figure 3.12 show the simulation time in seconds under fast-forwarding. We observe that BLRL achieves a substantial simulation time reduction compared to full warmup. NSL-BLRL reduces the overall simulation time even further, even to a level where warmup using NSL-BLRL is nearly as fast as no-warmup. In other words, the cost for warming up the microarchitecture state under fast-forwarding is nearly zero under NSL-BLRL. Note also that different percentiles $K\%$ have limited effect on the overall simulation time. We can conclude that a percentile $K = 100\%$ is the optimal choice since it gives the highest accuracy while incurring no additional simulation time overhead compared to smaller percentiles $K\%$.

Checkpointing

The simulation times for checkpointed simulation are presented in Figure 3.13. BLRL yields substantial simulation time reductions over full warmup. Note that the simulation time reductions under checkpointing are even bigger than under fast-forwarding. This is to be ex-

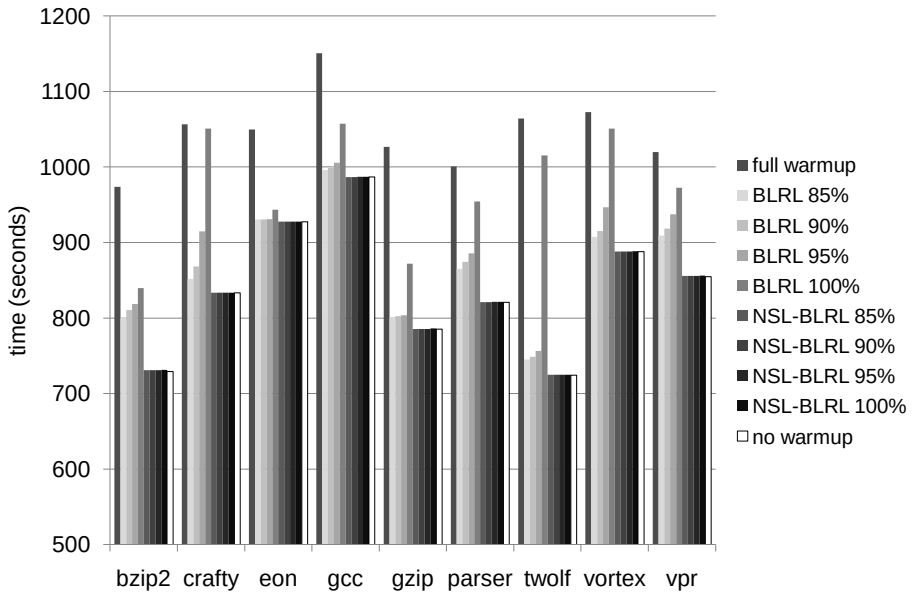

Figure 3.12: Simulation time for BLRL and NSL-BLRL for sampled simulation using fast-forwarding.

pected as checkpointed simulation does not require simulating the pre-sampling unit as opposed to fast-forwarding. Another interesting note is that the simulation time reduction when comparing NSL-BLRL versus BLRL under checkpointing is higher than under fast-forwarding. Under fast-forwarding, NSL-BLRL achieves a reduction in simulation time over BLRL up to a factor 1.4×; under checkpointing, NSL-BLRL achieves a 2.9× to 14.9× simulation time speedup over BLRL. This is to be explained for the same reason as detailed earlier; checkpointed simulation does not involve functional simulation.

3.5.4 Storage requirements

We now quantify the storage requirements of NSL-BLRL for storing the cache state checkpoints on disk. Figure 3.14 shows the amount of storage requirements for NSL-BLRL compared to NSL — BLRL does not require any significant storage. The numbers shown in Figure 3.14 represent the amount of storage (in MB) needed to store one cache state checkpoint in compressed format. For NSL, the average compressed storage requirement per sampling unit is 810 KB; the maxi-

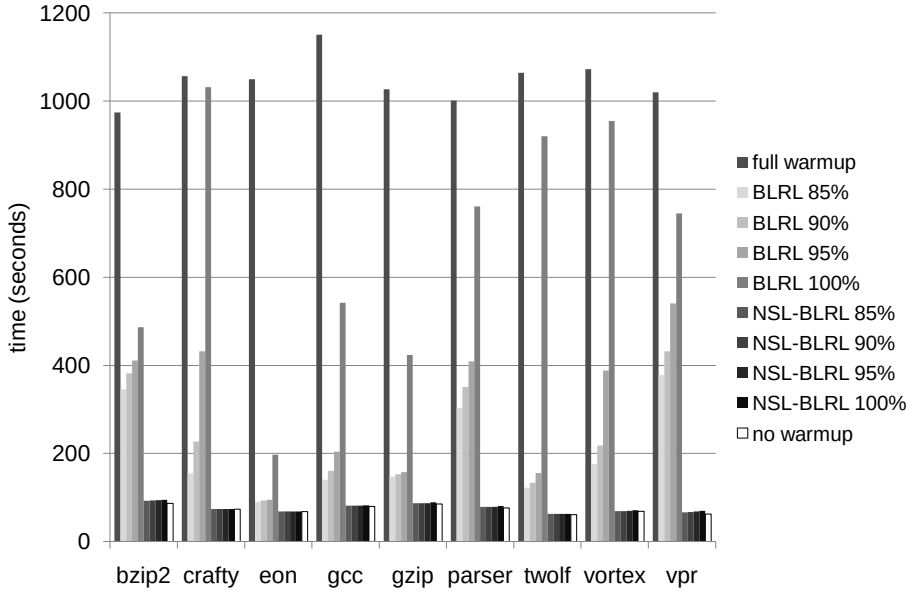

Figure 3.13: Simulation time for BLRL and NSL-BLRL for sampled simulation using checkpointing.

mum observed is for `bzip2`: 2.5 MB. The storage requirements are significantly smaller for NSL-BLRL compared to NSL. For example, for $K = 100\%$, the average storage requirement is 553 KB (a 32% reduction); for $K = 95\%$, the average storage requirement is 425 KB (a 48% reduction). We thus conclude that the real benefit of NSL-BLRL compared to NSL is its reduced storage requirements — NSL-BLRL and NSL are comparable in terms of accuracy and simulation time. In case a larger number of checkpoints need to be stored on disk for a complete benchmark suite, then we can easily end up with thousands of samples and respective checkpoint files. For example, for SimPoint there are 7392 1 M instruction samples for the entire SPEC CPU2000 benchmark suite[8]. If 810 KB needs to be stored on disk per sampling unit, then approximately 6 GB disk space is required for storing the NSL cache state warmup info. Note that this is an optimistic approximation. In our experimental setup we assumed 100 M instruction pre-sampling units. Larger pre-sampling units will result in even larger NSL warmup checkpoints to be stored on disk, as discussed previously (see also Figure 3.11). Hence, the total storage requirements are expected to be sub-

[8] `http://www.cs.ucsd.edu/~calder/simpoint`

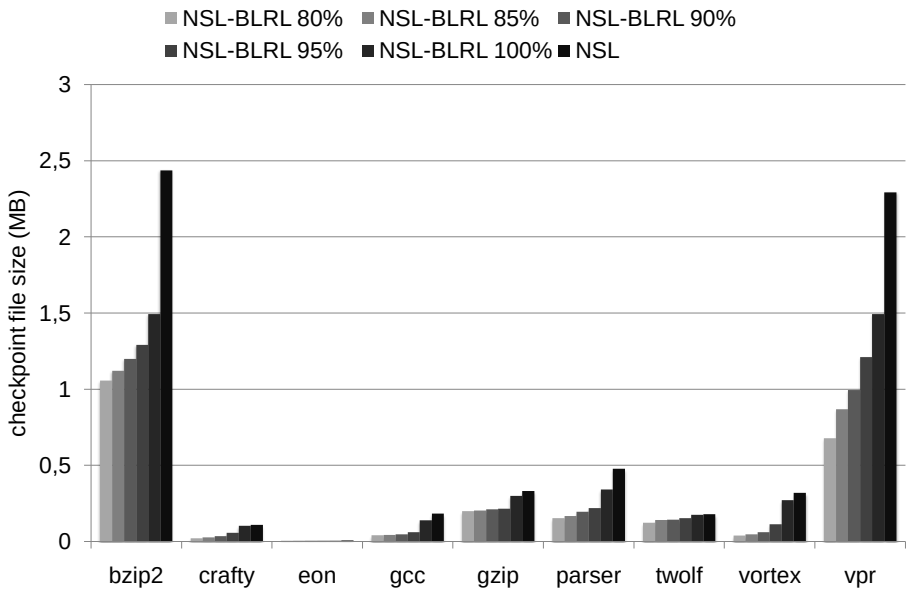

Figure 3.14: Storage requirements for NSL-BLRL compared to NSL: average number of MBs of disk storage needed for storing one MSI checkpoint in compressed format.

stantially larger than the 6 GB mentioned above. In addition, the ASI needs to be stored on disk as well. Even though storage is cheap these days, maintaining such large checkpoint files might be impractical to do. We conclude that NSL-BLRL is capable of reducing the total disk space requirements for MSI checkpointing by approximately 30% without any loss in accuracy.

3.5.5 Cache replacement policies

NSL achieves perfect warmup for LRU caches, by construction; however, it is unclear whether NSL-BLRL is an accurate technique for warming caches under different cache replacement policies. This is evaluated in Figure 3.15, which compares the IPC between continuous warmup and NSL-BLRL 100% for the FIFO, random and LRU replacement policies. The IPC prediction error increases slightly for the FIFO and random replacement policies compared to LRU. The average prediction error for LRU is 0.3% whereas the average prediction errors for FIFO and random are 1.3% and 2.3%, respectively. This is to be expected because NSL only guarantees perfect warmup for caches with

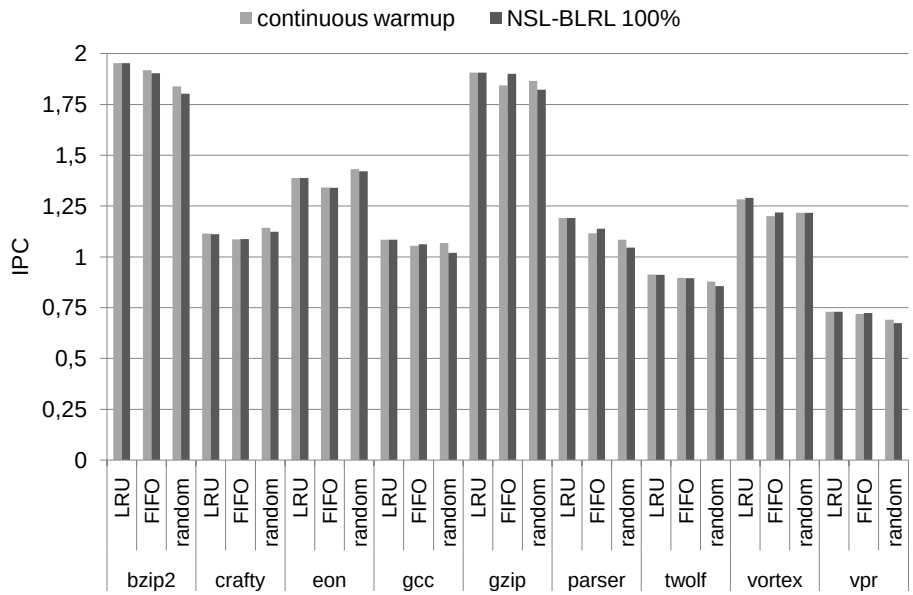

Figure 3.15: IPC for continuous warmup and NSL-BLRL 100% for different cache replacement policies.

an LRU replacement policy.

3.6 Summary

Computer architecture research and development relies heavily on sampling for speeding up architectural simulation. The idea of sampled simulation is to simulate only a small fraction of a benchmark's dynamic instruction stream (the sample). However, there are three major challenges for sampled simulation to be accurate and fast: (i) the selection of representative sampling units, (ii) initializing the sampling unit's architecture state, and (iii) estimating the sampling unit's microarchitecture state, also referred to as the cold-start problem.

This chapter addressed the cold-start problem by combining No-State-Loss (NSL) with Boundary Line Reuse Latency (BLRL) into a new cache warmup strategy: NSL-BLRL. The basic idea is to truncate the NSL stream of memory references in a pre-sampling unit using BLRL information. The NSL stream is the least recently used sequence of memory references in the pre-sampling unit. BLRL then selects a fraction of this NSL stream based on how far back warmup needs to go in

the pre-sampling unit to accurately warmup the microarchitecture state for the given sampling unit. The NSL-BLRL warmup info could then be viewed as a microarchitecture state checkpoint. Warming up a cache hierarchy using NSL-BLRL is then done by loading the checkpoint from disk and warming the caches using the NSL-BLRL reference stream. Compared to other existing microarchitecture state checkpointing techniques, NSL-BLRL is more flexible in the sense that the warmup info can be used for a broader range of hardware configurations. For example, whereas Memory Hierarchy State (MHS) and the TurboSMARTS' live-points approaches require a fixed cache block size, NSL-BLRL does not.

We showed that NSL-BLRL is substantially faster than BLRL, i.e., the number of warmup instructions is reduced by up to three orders of magnitude. Also, NSL-BLRL is nearly as accurate as BLRL — the small deviation is mainly because of not modeling dirty cache lines in NSL-BLRL. The shorter warmup length for NSL-BLRL results in substantial simulation speedups against BLRL. Under fast-forwarding, the simulation speedup is up to 1.4×, while under checkpointing, the simulation speedup ranges between 2.9× and 14.9×. Compared to NSL, the benefit of NSL-BLRL is in the reduced checkpoint files that need to be stored on disk. NSL-BLRL typically yields 30% smaller MSI checkpoints which is important when it comes to storing a large number of checkpoint files on disk for a large number of sampling units. In addition, reducing the checkpoint size allows for a faster checkpoint distribution across a cluster of machines in case of parallel sampled simulation.

Chapter 4

Benchmark synthesis

Nothing can be created from nothing.
Lucretius, De Rerum Natura

Code mutation conceals the proprietary information of an application to facilitate benchmark sharing, while, on the other hand, sampled simulation is very effective at reducing simulation time. However, extending both approaches to enable compiler and ISA exploration is non-trivial.

This chapter presents a benchmark synthesis framework that generates short-running benchmarks to limit simulation time; in addition, because the benchmarks are synthetically generated from a number of program characteristics, they do not reveal proprietary information. To enable architecture and compiler exploration, we generate the benchmarks in a high-level programming language. We extensively discuss the benchmark synthesis framework along with a number of possible applications.

4.1 Introduction

As mentioned before, code mutation and sampled simulation can be used in cascade to generate short-running benchmarks that are representative of (proprietary) applications and that can be distributed to third parties without revealing intellectual property. Although this is a feasible approach, there are a number of limitations. For one, while the top-down approach of code mutation yields highly representative benchmarks, it lacks flexibility. More specifically, it does not allow for altering workload behavior easily. For example, for the generation of

future workloads, a bottom-up approach would be more suitable, i.e., one could model performance characteristics that are to be expected for future applications into a synthetic benchmark. The importance of anticipating future application behavior is shown in [Yi et al., 2006]. This study shows that the performance of a processor optimized for the SPEC CPU95 benchmarks is more than 20% slower than the performance of a processor optimized for the SPEC CPU2000 benchmarks, when executing the SPEC CPU2000 benchmarks.

Second, code mutation acts at the binary level, i.e., it mutates a binary into a benchmark mutant through binary rewriting. Consequently, a benchmark mutant can only be used for exploring the microarchitectural space. To explore the architectural and compiler space as well, a benchmark mutant needs to be generated for each ISA/compiler. A similar problem arises when using sampled simulation during architecture exploration; comparing multiple binaries of the same source code requires identifying sampling units that represent the same behavior across these binaries [Perelman et al., 2007].

Finally, sampled simulation requires simulator support to establish the architecture state (to go from one sampling unit to the next) and the microarchitecture state (to minimize the cold-start problem) at the beginning of a sampling unit. Also, sampling units can be executed on functional and performance simulators, but not on hardware. This implies that sampled simulation is not ideal for verifying that a performance model is accurate with respect to hardware (in the literature referred to as 'performance model validation'). The underlying reason is that sampled simulation introduces an additional error — next to the error from an inaccurate performance model — which complicates the calibration of the performance model.

In this chapter, we propose a novel benchmark synthesis approach [Van Ertvelde and Eeckhout, 2010a] that aims at addressing these additional limitations. More specifically, we present a benchmark synthesis framework with three key features. First, it generates synthetic benchmarks in a high-level programming language (HLL) to enable the exploration of both the architecture and compiler spaces, in contrast to prior work in benchmark synthesis which generates synthetic benchmarks in assembly. Second, the synthetic benchmarks hide proprietary information from the original applications they are built after. Hence, companies may want to distribute synthetic benchmark clones to third parties as proxies for their proprietary codes; third parties can

then optimize the target system without having access to the original codes. Third, the synthetic benchmarks are short-running compared to the original applications they are modeled after, yet they are representative. The framework involves two key steps: (i) profiling the real-world (proprietary) application to measure its execution characteristics, and (ii) generating a synthetic benchmark clone in a high-level language (C in our case) based on this execution profile.

4.2 High-level language benchmark synthesis

Our goal is to generate a synthetic benchmark in a high-level programming language that is similar to a real application in terms of its execution behavior across architectures and compilers, yet it should not expose proprietary information and it should be short-running compared to the real application. This is a non-trivial problem to solve. We now describe how we approach this problem at a high level — we discuss the various steps of our benchmark synthesis framework in greater depth in Section 4.3.

4.2.1 Framework overview

Figure 4.1 provides a high-level view of the overall framework. We start from a real-world application. This could be a proprietary application with a proprietary input. This application is then compiled at a low optimization level, e.g., `-O0` in GNU's GCC. The reason for doing so is to facilitate pattern recognition and to enable compiler research, as we will explain later on. We then run the resulting binary and profile its execution, i.e, we count how many times a loop is iterated, how often a basic block is executed, etc. — this information is stored in a novel structure: the 'Statistical Flow Graph with Loop information' (SFGL). In addition, we record memory access patterns and branch taken and transition rates. Finally, we employ a (simple) pattern recognizer that scans the executed code to identify C code statements that correspond to sequences of instructions observed at the binary level. This pattern recognizer translates the binary code to C code in a semi-random fashion in order to obfuscate proprietary information.

All the characteristics that we collect are comprised in a statistical profile that captures the execution behavior of the original application and its input. We then generate a synthetic benchmark from this sta-

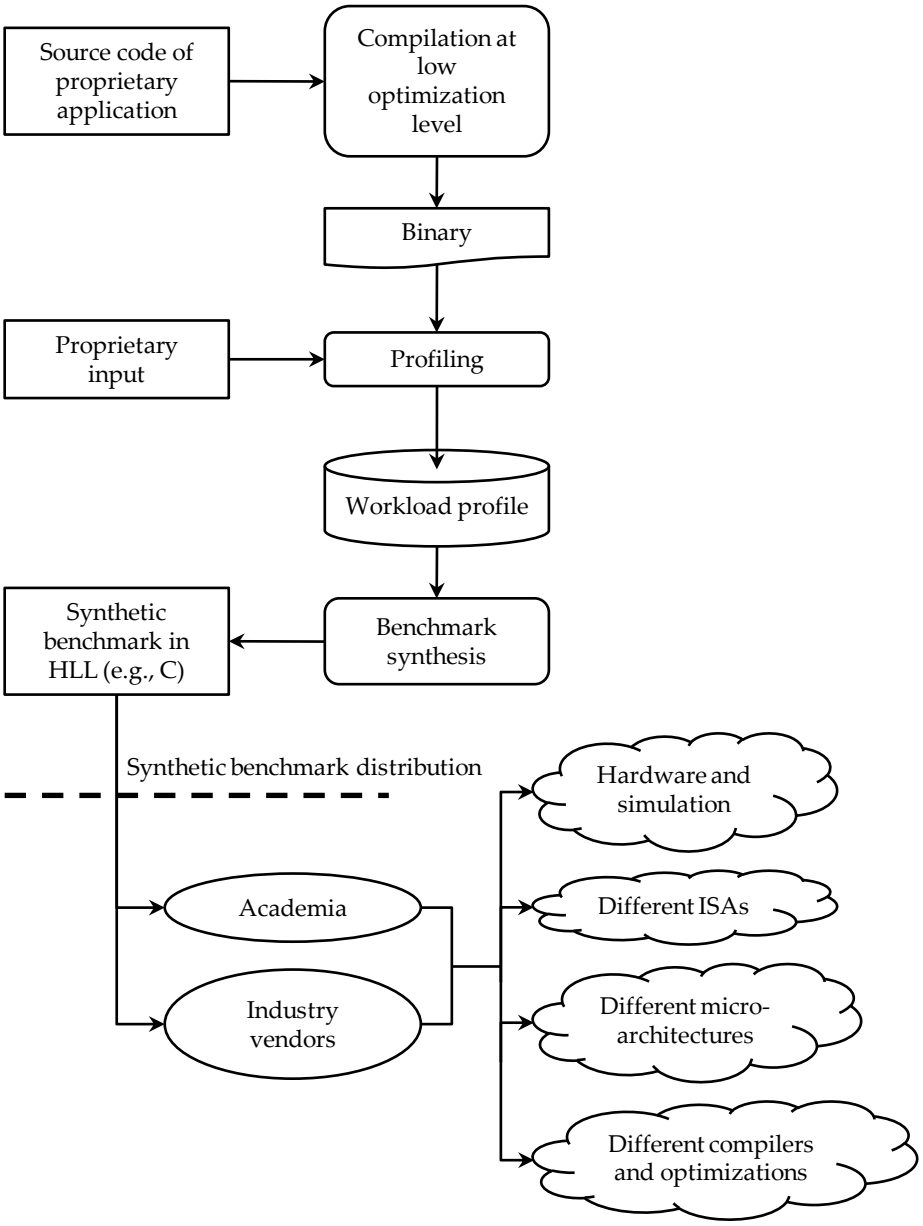

Figure 4.1: Benchmark synthesis framework overview.

tistical profile. This is done in a high-level programming language, in our case C: we generate sequences of C code statements (basic blocks), as well as if-then-else statements, loops and function calls, and we add inter-statement dependencies as well as data memory access patterns. The C code structures are generated pro rata their occurrences in the original application. However, we force the synthetic benchmark to execute fewer instructions than the original application, by construction. This is done by reducing the execution frequencies of basic blocks, loops and function calls by a given reduction factor R. The end result is a synthetic benchmark that executes fewer instructions than the original application while being representative of the original application.

The synthetic benchmark does not expose proprietary information (because of the semi-random binary to source code translator, and the workload reduction) and can thus be distributed to third parties. Because the synthetic benchmarks are generated in a high-level programming language, they enable exploring the architecture and compiler space, and comparing systems with different compilers, and optimization levels, as well as different instruction-set architectures, microarchitectures and implementations. The synthetic benchmarks can run on execution-driven simulators as well as on real hardware.

A important aspect of our approach is that we compile the original application at a low compiler optimization level before profiling. The reason for doing so is to force the compiler not to perform aggressive optimizations. This facilitates the pattern recognition and translation from binary code to C code, and, more importantly, it enables generating synthetic benchmarks that can be used to explore the compiler space, as we will demonstrate in the evaluation section.

Before describing the different steps of our framework in more detail, we discuss the potential applications of high-level language benchmark synthesis.

4.2.2 Applications

We believe the proposed framework has a number of potential applications:

- **Distributing synthetic benchmarks as proxies for proprietary applications.** The most obvious application is to use the framework to generate synthetic clones for real-world proprietary applications. There are many possible application scenarios, both in

the embedded and server/datacenter spaces. For example, phone companies may not be willing to share their proprietary software with a processor vendor in order to optimize the processor architecture for the next-generation cell phone, yet they may be willing to share a synthetic clone. A similar application scenario applies to service providers in the cloud: they will be reluctant to share their platform software, yet they may want to distribute synthetic clones to third-party hardware vendors. The same applies to compiler builders: they could evaluate their compiler performance based on the synthetic clones rather than the real applications. Of course, co-optimization of hardware and software, which is an important focus today given the emphasis on energy-efficient computing, can also rely on synthetic benchmark clones.

- **Simulation time reduction.** As mentioned earlier, the synthetic benchmarks are short-running compared to the original applications, i.e., their dynamic instruction count is significantly smaller. Because simulation time is an important concern in architecture research and development, benchmark synthesis also helps in reducing simulation time, and eventually the overall time-to-market. This is also important in the compiler space: for example, iterative compilation evaluates a very large number of compiler optimizations in order to find the optimum compiler optimizations for a given program [Cooper et al., 1999] [Kulkarni et al., 2004]. A synthetic clone that executes faster could reduce the overall compiler space exploration time.

- **Generate emerging workloads.** The framework can also be used to generate emerging and future workloads. In particular, one can generate a statistical profile with performance characteristics that are to be expected for future emerging workloads. For example, one could generate specific sequences of C statements, a particular memory access behavior (e.g., large working set, random access patterns), etc. The synthetic benchmarks generated from the profiles can then be used to explore design alternatives for future computer systems.

- **Model hard-to-setup workloads.** Similarly, one could build proxy benchmarks for workloads that are hard to setup. For example, database workloads and commercial workloads in general are non-trivial to setup [Shao et al., 2005]. Synthetics could

be a way to facilitate the benchmarking process using commercial workloads. In fact, an additional advantage of generating synthetic benchmarks in a high-level programming language compared to assembly synthetic benchmarks is that interfacing libraries can be done easily using existing APIs.

- **Benchmark consolidation.** Multiple applications can also be consolidated into a single synthetic benchmark. Basically, by putting together the statistical profiles from different applications, one can generate a single consolidated synthetic benchmark that is representative of a set of applications. Benchmark consolidation also helps hiding and obfuscating proprietary information.

4.3 Framework details

We now describe the two key steps of our benchmark synthesis framework in more detail: (i) collecting the execution profile, and (ii) generating a synthetic benchmark clone based on this profile.

4.3.1 Collecting the execution profile

The profiler collects a number of execution characteristics, which we discuss in the following subsections.

Statistical Flow Graph with Loop annotation (SFGL)

The central structure in the statistical profile is the SFGL which captures a program's control flow behavior in a statistical manner. Figure 4.2 shows an example. The nodes represent basic blocks and the edges represent control flow transitions. Each node is annotated with the basic block's execution count, and each edge is annotated with the respective transition probability. For example, basic block A executes 10 times, and is followed nine times out of ten by basic block B, and one time out of ten by basic block C. Basic block D on the other hand, is always followed by basic block E.

The SFGL also identifies the loops along with the number of iterations that each loop executes. We use the loop detection algorithm by Aho et al. [2006]. For example, E→J is a loop comprising basic blocks E, F, G, H, I and J. The explanation is that (i) there is no way to reach basic

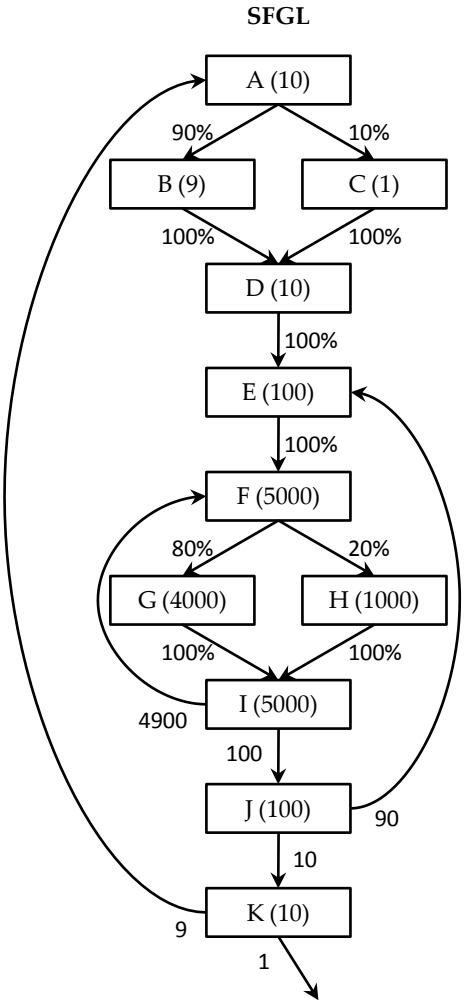

Figure 4.2: An example SFGL (Statistical Flow Graph with Loop annotation).

block J without going through basic block E, and (ii) there exists a back edge that closes the loop [Aho et al., 2006].

For each instruction in each basic block we also record its instruction type. We consider a number of instruction types such as addition, subtraction, multiply, divide, branches, etc., and we make a distinction between integer and floating-point instructions. We also keep track of the instruction's input operands (constant, register, memory) and output operand (register or memory).

Branch taken and transition rate

For each conditional branch that is not a loop back edge, we determine its taken and transition rate. The branch taken rate is defined as the fraction of taken outcomes per branch, while the branch transition rate is defined as the number of times a branch switches between taken and not-taken during execution [Haungs et al., 2000]. A low transition rate means that the branch is either mostly taken or mostly not taken, and a high transition rate means that the branch constantly changes between taken and not-taken. High and low transition rates typically suggest easy to predict branches. A medium transition rate suggests hard to predict branches.

For example, if the branch sequence at the end of basic block A in Figure 4.2 is '1111111110' (whereas 1 means 'taken'), the branch transition rate is 10% and the taken rate is 90%. Both the branch taken rate and transition rate are independent of a particular branch predictor. We classify branches into two classes, easy to predict branches (branches with a transition rate less than or equal to 10% or greater than or equal to 90%) and hard to predict branches. The branch at the end of basic block A is thus classified as easy to predict.

Memory access patterns

For each memory access we record its cache hit/miss ratio. We do this by simulating a cache structure in the profiling tool — it is possible to compute cache miss rates across a range of cache organizations in a single pass [Hill and Smith, 1989]; this involves computing LRU stack distances to model temporal locality and memory address distances to model spatial locality (see Section 3.3 in Chapter 3 for how to build and maintain an LRU stack).

Table 4.1: Memory access strides for generating a target miss rate (assuming a 32-byte cache line and a 32-bit architecture).

Class	Miss rate range	Stride (bytes)
0	0% - 6.25%	0
1	6.25% - 18.75%	4
2	18.75% - 31.25%	8
3	31.25% - 43.75%	12
4	43.75% - 56.25%	16
5	56.25% - 68.75%	20
6	68.75% - 81.25%	24
7	81.25% - 93.75%	28
8	93.75% - 100%	32

We classify the memory accesses in a number of classes according to their hit/miss ratios, see Table 4.1 . Data memory accesses are modeled using these simple stream access classes, as we will describe later.

4.3.2 Synthetic benchmark generation

The second step in our HLL benchmark synthesis framework is to generate a synthetic benchmark by modeling all the workload characteristics described in the previous subsections into a synthetic clone. This generation process is done in three sub-steps: (i) SFGL downscaling, (ii) generation of basic blocks, loops and C-functions, and (iii) C code generation.

Scale down the SFGL

We first compute a downscaled SFGL by reducing the occurrences of the basic blocks and loop counts in the SFGL. This is done by dividing the basic block execution counts and loop iteration counts by a reduction factor R. For nested loops, we first scale the iteration count of the outer loop. If the iteration count of the outer loop is smaller than the reduction factor, we also downscale the nested loop, etc. Basic blocks and loops that are executed infrequently (i.e., less than R times) are removed from the SFGL. Along with this removal, we also remove all incoming and outgoing edges. The purpose for downscaling is twofold:

(i) generate short-running synthetic benchmarks, and (ii) obfuscate the original application's semantics.

Figure 4.3(a) shows the downscaled SFGL of the example shown in Figure 4.2. The reduction factor equals 2 in this example[1]. Basic block C (along with its incoming and outgoing edges) does no longer appear in the downscaled SFGL. Also, the iteration count of the outer loop (A→K) is scaled down from 10 to 5. Note however that the (average) iteration count of loop E→J remains 10 ($(45/5) + 1$). Figure 4.3(b) shows the downscaled SFGL for a reduction factor of 50. The outer loop (A→K) does no longer appear. For a reduction factor of 200, loop E→J also gets removed, see Figure 4.3(c), and loop F→I gets scaled down to 25 iterations.

Generate basic blocks and loops

Once the downscaled SFGL is computed, we start generating the skeleton for the synthetic benchmark. We first pick a (random) basic block. If this basic block is part of a loop, we generate the loop that contains this basic block; for example, if basic block F would be picked in Figure 4.3(c), the framework would generate a loop comprising basic blocks F, G, H and I. If the loop itself is nested in a bigger loop, we first generate the outer loop and then generate the inner loops. If the basic block is not part of a loop, we determine its successor(s) and start building the control flow structure of the synthetic benchmark. If there are no successors to a basic block (because the successor basic blocks got removed during down-scaling), we re-start the generation algorithm and pick a random basic block. For each basic block and loop that we generate, we decrease the respective execution counts to reflect the fact that these basic blocks and loops have been generated. Basic blocks and loops with zero execution counts are removed from the SFGL. We continue this process until all basic blocks in the SFGL have been selected and the SFGL is empty.

Formally, this corresponds to the following algorithm:

1. Terminate the algorithm when all the basic blocks (and loops) have been selected and the downscaled SFGL is empty. Otherwise, select a (random) basic block.

[1]Note that we use a small reduction factor for illustrative purposes. For the applications considered in the evaluation section, we typically use larger reduction factors.

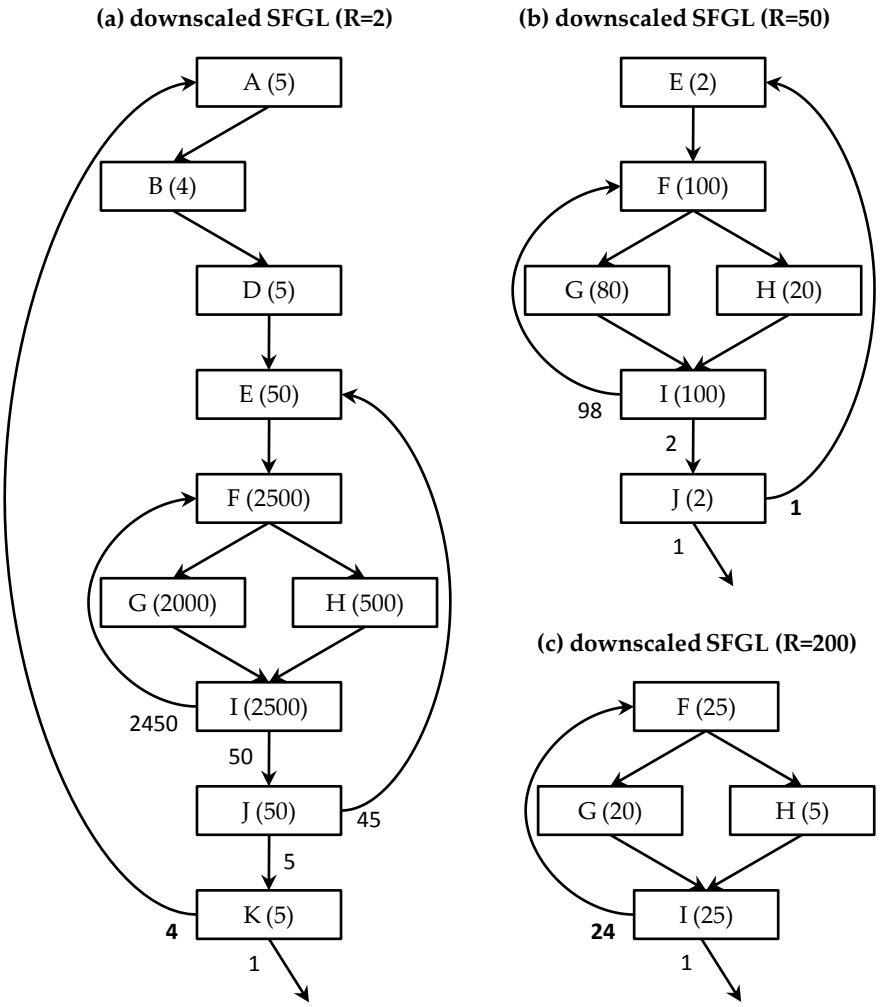

Figure 4.3: Downscaled SFGL with a reduction factor R = 2, 50 and 200, compared to the original SFGL shown in Figure 4.2.

2. If this basic block is part of a loop, determine the outer loop that contains this basic block (we first generate outer loops and then generate the inner loops). Otherwise go to step 4.

3. Generate this loop with the iteration count the number of times the loop needs to be iterated according to the downscaled SFGL. Reduce the execution count of this loop in the SFGL with the number of iterations so as to reflect the fact that this loop has been generated. If the execution count is zero, remove this loop from the SFGL. Finally, select the first basic block (of this loop) that needs to be generated.

4. If this basic block is the beginning of a loop, goto step 3. If this basic block is the beginning of an if-then-else statement, goto step 5. Otherwise goto step 6.

5. First generate the basic blocks following the `if-then` statement, and subsequently generate the basic blocks following the `else` statement. Select the first basic block of the `then/else` statement.

6. Generate this basic block and decrease the execution count to reflect the fact that this basic block has been generated — take the number of iteration counts into account.

7. Determine its successor(s) and goto step 4. If there are no successors[2] to a basic block (because the successor basic blocks got removed during down-scaling), goto step 1.

When we apply this algorithm to the downscaled SFGLs from Figure 4.3, we get the skeleton code shown in Figure 4.4. For example, consider the downscaled SFGL shown in Figure 4.3(b). Following the algorithm, we first pick a (random) basic block, e.g., basic block G. This basic block is part of loop F→I, and loop F→I is in its turn part of outer loop E→J. Hence, we first generate the outer loop E→J, which is downscaled to 2 (100/50) iterations. This translates to `for(j=0; j<2; j++){}` in C code, as also shown in the lower left corner of Figure 4.4. We then determine the successor of basic block E, which is basic block F, and we generate loop F→I: `for(k=0; k<50; j++){}`. Basic

[2]For the sake of clarity, this algorithm does not cover all the possible exceptions, e.g., when there are no successors to a BBL in a loop, we select a random BBL in the same loop (if any)

Original skeleton code, see Figure 4.2

```
for(i=0;i<10;i++){
  if(){
  }
  else{
  }
  for(j=0;j<10;j++){
    for(k=0;k<50;k++){
      if(){
      }
      else{
      }
    }
  }
}
```

Skeleton code for Figure 4.3(a), R=2

```
for(i=0;i<5;i++){
  if(){
  }
  else{
  }
  for(j=0;j<10;j++){
    for(k=0;k<50;k++){
      if(){
      }
      else{
      }
    }
  }
}
```

Skeleton code for Figure 4.3(b), R=50

```
for(j=0;j<2;j++){
  for(k=0;k<50;k++){
    if(){
    }
    else{
    }
  }
}
```

Skeleton code for Figure 4.3(c), R=200

```
for(k=0;k<25;k++){
  if(){
  }
  else{
  }
}
```

Figure 4.4: The upper left figure shows the skeleton code for the original application. The other figures show the skeleton code for the downscaled SFGLs.

block F is also the beginning of an if-then-else statement; after generating this construction, the downscaled SFGL is empty and the algorithm terminates. The end result is shown in Figure 4.4, along with the skeleton codes for the other downscaled SFGLs. These examples show how we first scale the iteration count of the outer loop, and, if necessary, then scale the iteration count of the nested loop(s).

Function assignment

We subsequently organize the basic blocks and loops into C-functions. This organization does not necessarily correspond to the C-functions observed in the original application — again, this is to hide proprietary information in the synthetic benchmark.

Generate C statements

Once we have the skeleton synthetic benchmark consisting of C-functions, loops and basic blocks, we now populate the basic blocks with C statements. This is done by scanning the instruction types of all the instructions in each basic block, and identifying C statements that correspond to these sequences of instructions. Table 4.2 shows the most important patterns and how they are translated into C statements. The patterns cover over 95% of the instructions for all the benchmarks. Coverage is not 100% (which again helps in hiding proprietary information): to compensate for the uncovered instructions we keep track of the number of operations and types that have been translated so far, and we compensate for those instructions on a later occasion. For example, if we are lagging behind in the number of loads, we try to generate a 'load-load-arithmetic-store' instead of a 'load-arithmetic-store' pattern. Or, if we are lagging behind in the number of stores, we will generate an additional 'store' pattern.

When reaching the end of a basic block we generate a branch statement. This can be either a loop back edge or a conditional branch. In case of a loop back edge, we generate a `for` loop with the iteration count the number of times the loop needs to be iterated according to the downscaled SFGL, as mentioned before.

Table 4.2: Generating C statements through pattern recognition. The `op` refers to an operation (e.g., addition, subtraction, etc.); the `cst` refers to a randomly generated constant value.

Pattern	Example	C statement
load-store	`movl t+512,%eax` `movl %eax,t+504`	`mem[i] = mem[j];`

load-arithmetic-store	`movl t+512,%eax`
	`addl $2,%eax`
	`movl %eax,t+504`
		`mem[i] = mem[j]`
		`op cst;`

load-load-arith-store	`movl t+508,%edx`
	`movl t+512,%eax`
	`leal (%edx,%eax)`
	`,%eax`
	`movl %eax,t+504`
		`mem[i] = mem[j]`
		`op mem[k];`

load-load-arith-load-reg-arith-reg-store	`movl t+508,%edx`
	`movl t+512,%eax`
	`addl %eax,%edx`
	`movl t+516,%eax`
	`movl %edx,%ecs`
	`subl %eax,%ecx`
	`movl %ecx,%eax`
	`movl %eax,t+504`
		`mem[i] = mem[j] op`
		`mem[k] op mem[l];`

load-compare-branch	`movl t+504,%eax`
	`cmpl $3,%eax`
	`jbe`
		`if(mem[i] > cst)`

store	`movl $9,%eax`	`mem[i] = cst;`

For a non-loop branch, we generate an if-then-else statement, and we make a distinction between branches with a low transition rate ($\leq$10%), a medium transition rate (10%$<$ rate $<$90%), and a high transition rate ($\geq$90%). Branches with a low transition rate are modeled to match their taken rates using a compare operation on a loop iterator. Consider, for example, the skeleton code shown lower right in Figure 4.4; the `if-then` path is taken 20 times, and the `else` path is taken 5 times. Assuming a low transition rate, this translates to the following C code:

```
for(k=0;k<25;k++){
  if(k<20){
  }
  else{
  }
}
```

Obviously, a branch predictor should be able to predict the outcome of this conditional branch accurately.

If a branch is always (not-)taken, we fill the non-executed path with C statements that print out the results that have been computed elsewhere in the synthetic benchmark — this is to force the compiler not to optimize code away that is needed to preserve representativeness while producing data that is never used.

Branches with a high transition rate on the other hand, are modeled using a modulo-2 operation on the iterator of its innermost outerloop. Hence, the branch transitions between taken and not-taken (note that such a branch is easy to predict for a history predictor but hard to predict for a bimodal predictor).

Finally, branches with a medium transition rate jump in one or the other direction based on their taken and transition rates, i.e., we model them to match their taken and transition rates using both a compare and a modulo operation. For example, consider the following branch sequence: '1111011010'. The taken rate is 70% and the transition rate is 50%. This is modeled as follows:

```
for(k=0;k<10;k++){
  if(k<5 || (k%2)==0){
  }
  else{
```

```
    }
}
```

One can verify that the branch sequence in this code example equals '1111101010'; the taken rate is 70% and the transition rate is 50%.

We also generate memory access streams. This is done by generating stride patterns for all memory accesses, following prior work by Bell and John [2005]. These patterns walk through pre-allocated memory with a particular stride; the stride value is determined by the memory access hit/miss ratio. Different hit/miss ratios lead to different stride values. For example, an always hit memory access is modeled through a zero stride. A 50% hit rate is modeled through a stride value of 4; this will lead to a 50% miss rate assuming a 32 byte cache line size and a 32-bit machine, see also Figure 4.5.

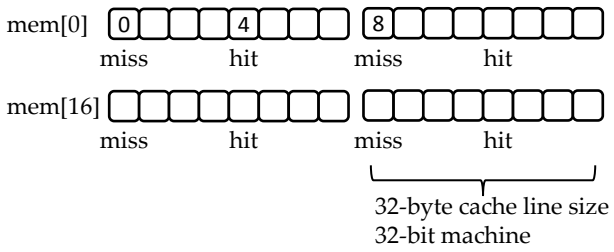

Figure 4.5: Generating stride patterns. A stride value of 4 results in a 50% hit rate.

4.3.3 Example

The code snippet below shows the Fibonacci kernel along with the automatically generated synthetic clone. The profiling was done with a particular input that calculates `fib(20)`, and we used a reduction factor of 2. This is reflected in the number of iterations that the loop is taking: the synthetic benchmark takes 10 iterations.

That specific input never caused an overflow, hence the `if` statement in the loop is never executed. The branches in both the original Fibonacci kernel and the synthetic program clone are easy to predict which is important to preserve the original program's performance characteristics. This example also illustrates that the Fibonacci kernel is no longer recognizable in the synthetic clone because the data depen-

dencies between the statements are very different between the original program and its clone.

The original Fibonacci kernel:

```
int fib(int n){
  int a=0, b=1, i=0,
  sum=0;

  for(i=0;i<n;i++){
    sum = a + b;
    if(sum<0){
      printf("overflow");
      break;
    }
    a = b;
    b = sum;
  }
  return sum;
}
```

The generated synthetic program clone:

```
unsigned int mStream0[256];
int i=0,j=0;

int f(){
  for(i=0;i<10;i++){
    mStream0[4] = mStream0[7] + mStream0[2];
    if(mStream0[0]==0x99){
      for(j=0;j<256;j++) printf("%d;", mStream0[j]);
    }
    mStream0[6] = i;
    mStream0[7] = mStream0[6];
  }
}
```

4.3.4 Limitations

Before evaluating our benchmark synthesis approach, we first discuss the current limitations of our framework.

- Different program characteristics are modeled independently of each other — the framework currently takes a first-order approach and assumes that the characteristics are uncorrelated. For

example, memory access behavior is modeled independently of control flow behavior and its interaction is not modeled. This is obviously not the case in real programs. Modeling second-order effects is likely to improve accuracy.

- The memory access behavior is based on cache miss rates and hence it is specific to a particular memory hierarchy. Although it is possible to measure cache miss rates for a range of caches in a single run, as mentioned earlier, a better solution would be to have a microarchitecture-independent way of modeling memory access behavior. Further, the current approach assumes that memory accesses can be modeled using stride patterns. Future work though may focus on modeling less regular memory access patterns.

- The instruction-level parallelism model is simplistic in our current setup, as we assume random dependencies between instructions. A more accurate approach would be based on profiling information so that the distribution of data dependencies in the synthetic benchmark matches the original application — however, there is a trade-off in accuracy versus hiding proprietary information.

- The reduction factor is chosen empirically so that the synthetic benchmark executes approximately 10 million instructions, as we will describe later. A more accurate approach would base the reduction factor on how representative the synthetic benchmark is relative to the real application while taking into account phase behavior.

4.4 Experimental setup

The benchmarks used in this study are from the MiBench benchmark suite [Guthaus et al., 2001], see Table 4.3. Pofiling is done using Pin [Luk et al., 2005], which is a dynamic binary instrumentation tool. The cache simulations are done using Pin as well. The branch prediction results are obtained using PTLSim [Yourst, 2007]; we consider a hybrid branch predictor with a bimodal component along with a global

Table 4.3: Embedded benchmarks used for evaluating benchmark synthesis.

Benchmark	Description	Input
Automotive		
basicmath	mathematical calculations	small/large
bitcount	bit count algorithm	small/large
qsort	quick sort algorithm	large
susan	image recognition	small/large edges/corners/smoothing
Consumer		
jpeg	image (de)compression	large, encode
Office		
strinsearch	comparison algorithm	small/large
Network		
dijkstra	path calculation	small/large
patricia	network routing	small
Security		
sha	secure hash algorithm	small/large
Telecomm.		
ADPCM	Pulse Code Modulation	small/large, enc./dec.
CRC32	32-bit Cyclic Redundancy Check	small/large
FFT	Fast Fourier Transformation	small/large, FFT/IFFT
GSM	voice encoding/decoding	small/large, enc./dec.

Table 4.4: Machines used for evaluating benchmark synthesis.

Machine	ISA	Description
Itanium 2	IA64	Itanium 2 at 900 Mhz with 256 KB L2
Pentium 4, 2.8 GHz	x86	Pentium 4 at 2.8 GHz with 1 MB L2
Pentium 4, 3 GHz	x86	Pentium 4 at 3 GHz with 1 MB L2
Core 2	x86_64	Core 2 at 2.2 GHz with 2 MB L2
Core i7	x86_64	Core i7 at 2.67 Ghz with 8 MB L2

history-based component. We also run detailed cycle-accurate simulations using PTLSim and we simulate a 2-wide out-of-order processor.

We also run real hardware experiments on five machines, see Table 4.4. The machines include Pentium 4, Core 2, Core i7 and Itanium 2 processors, and three ISAs: x86, x86_64 and IA64.

We use GNU's GCC compiler v4.0.2 in all of our experiments for the x86 and x86_64 machines. For the Itanium 2 machine we use GCC v3.3.2. We consider four compiler optimization levels: `-O0`, `-O1`, `-O2` and `-O3`.

4.5 Evaluation

The evaluation of the framework is done in a number of steps. We evaluate whether the synthetic benchmarks correspond to the real applications with respect to their dynamic instruction count, instruction mix, cache performance, branch prediction behavior, and eventually overall performance across architectures and compilers. We also verify whether the synthetic benchmark clones hide proprietary information from the original applications using existing software plagiarism detection tools.

4.5.1 Performance characteristics

Dynamic instruction count

Figure 4.6 shows the reduction in dynamic instruction count between the synthetic benchmark and the original application. Recall that the synthetic benchmark generation process scales down the SFGL using a reduction factor. We choose the reduction factor such that the synthetic benchmark executes approximately 10 million instructions. This leads to a reduction factor ranging from 1 to 250. The fact that the reduction factor is low in a number of cases is because the MiBench benchmarks are fairly short-running, hence there is little simulation time reduction to be gained. On average though, we achieve a 30$\times$ reduction in dynamic instruction count.

Figure 4.7 shows the normalized dynamic instruction count across compiler optimization levels; we show average numbers here. The dynamic instruction count is an important optimization target for compilers, even on today's superscalar out-of-order processors [Eyerman

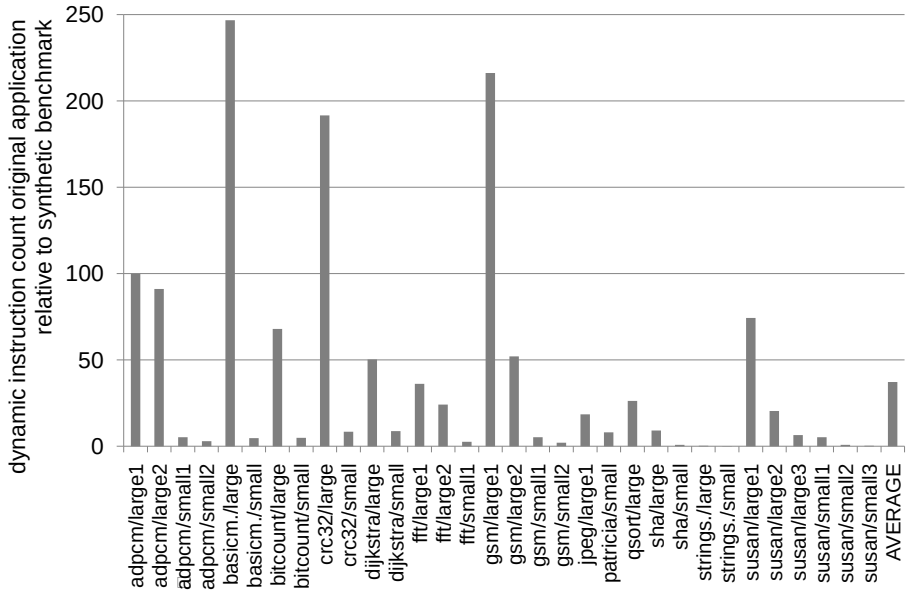

Figure 4.6: Reduction in dynamic instruction count.

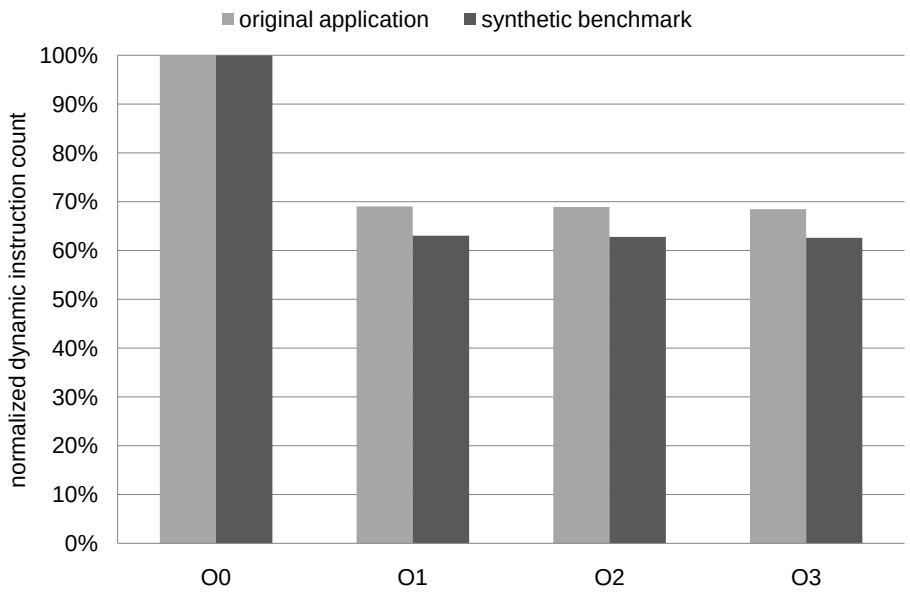

Figure 4.7: Normalized dynamic instruction count across compiler optimization levels.

et al., 2008]. The synthetic workload tracks the original workload fairly well: both suggest that the dynamic instruction count reduces by about a third when going from `-O0` to a higher optimization level.

Instruction mix, cache hit rates, and branch prediction behavior

Figures 4.8 and 4.9 show the instruction mix for the original applications and the synthetic benchmarks at the `-O0` and `-O2` optimization levels. Figures 4.10 and 4.11 show the data cache behavior for the original applications and the synthetic benchmarks at the `-O0` optimization level; Figures 4.12 and 4.13 show the data cache behavior at the `-O2` optimization level. Figure 4.14 shows results for the branch predictor behavior.

All of these graphs basically lead to the same conclusion. Although the synthetic benchmarks do not yield a perfect match with the original applications, they most often yield the same conclusions and insights. For example, both the synthetics and the real workloads see a decrease in the fraction of load instructions along with an increase in the fraction of arithmetic instructions at a higher optimization level, see the average bars on the righthand side in Figures 4.8 and 4.9; the reason is that optimizations such as copy propagation eliminate load instructions by replacing variables with their original values (if the variables do not change). In terms of data cache behavior, the synthetic benchmarks correlate well with the original applications, in spite of using a very simple cache access model. For example, `dijkstra` seems to be the benchmark that is most sensitive to cache space, see Figure 4.10, and a data cache size of 8 KB seems to capture most of the benchmark's working set (i.e., there is a significant increase in data cache hit rate going from 4 KB to 8 KB but a minor increase going from 8 KB to 16 KB). We observe the same trend for the synthetic version of `dijkstra`, see Figure 4.11.

Finally, for the branch predictor accuracy graph, see Figure 4.14, we observe that `adpcm` is most sensitive to the branch predictor; this is also captured by the synthetic benchmark. Also, for both the original and the synthetic workloads, `crc32` seems to be the program with the highest branch prediction accuracy; this is because the `crc32` kernel consists of one large loop with branches that have low transition rates. On the other hand, the synthetic branch prediction accuracy is significantly overestimated for `adpcm`, `jpeg`, `qsort` and `stringsearch`. The reason is that the current branch model is more oriented towards bimodal

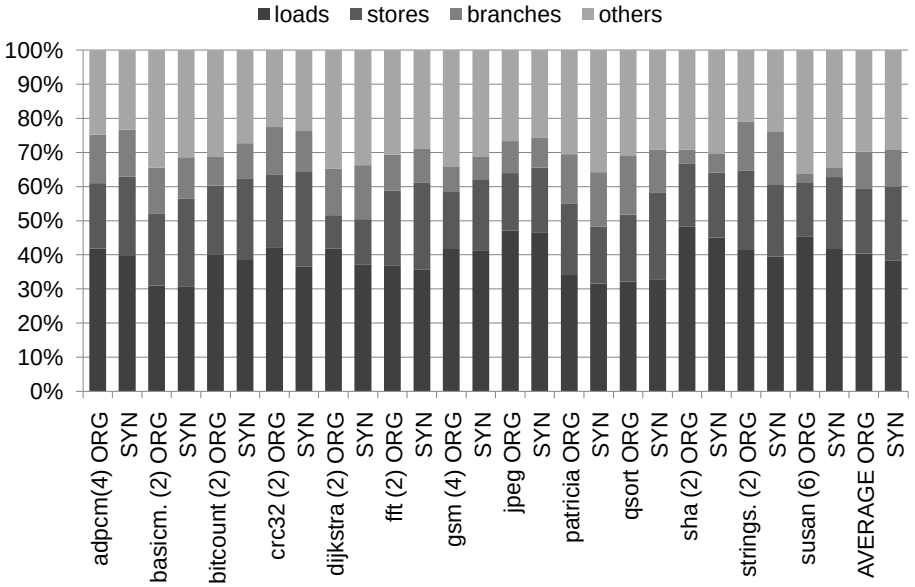

Figure 4.8: Instruction mix for the original applications (ORG) and the synthetic benchmarks (SYN) at the `-O0` optimization level.

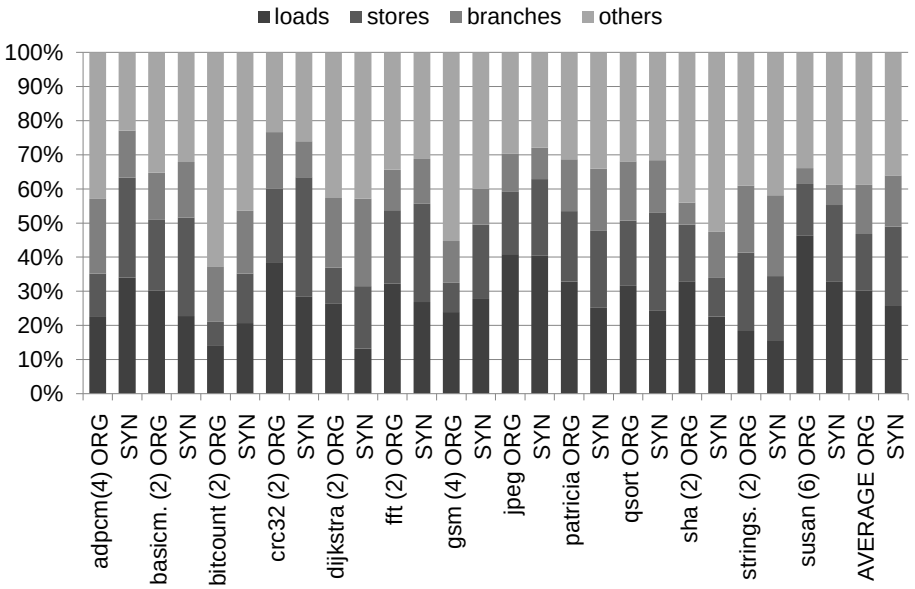

Figure 4.9: Instruction mix for the original applications (ORG) and the synthetic benchmarks (SYN) at the `-O2` optimization level.

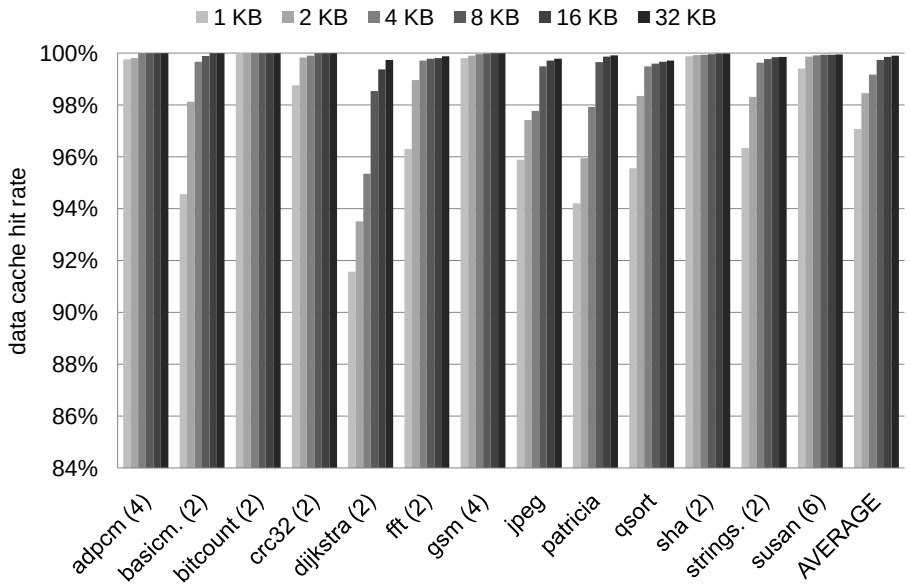

Figure 4.10: Data cache hit rates for the original applications at the `-O0` optimization level.

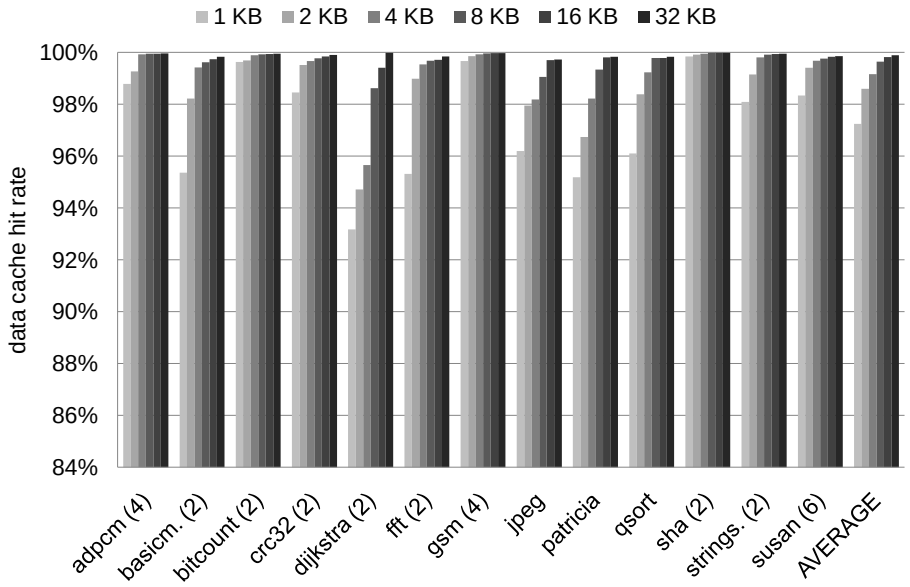

Figure 4.11: Data cache hit rates for the synthetic benchmarks at the `-O0` optimization level.

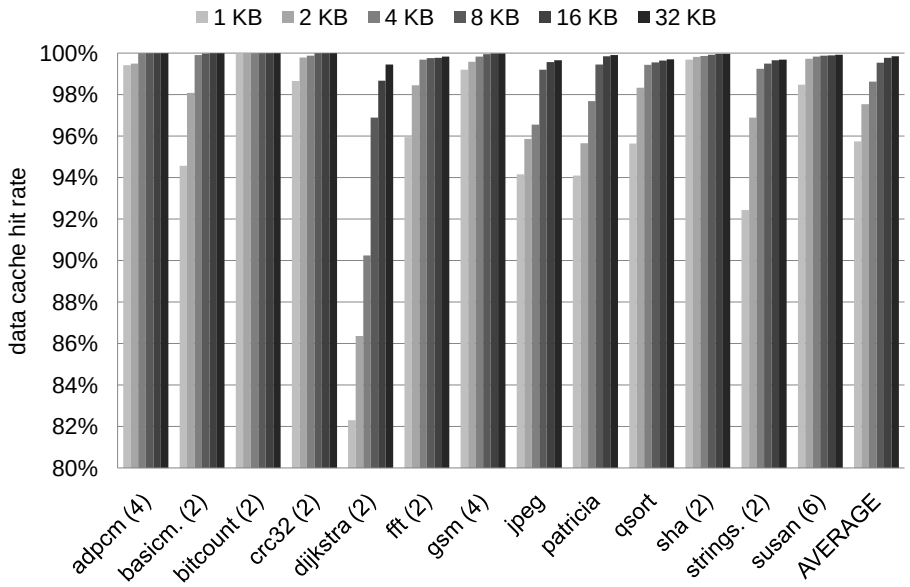

Figure 4.12: Data cache hit rates for the original applications at the `-O2` optimization level.

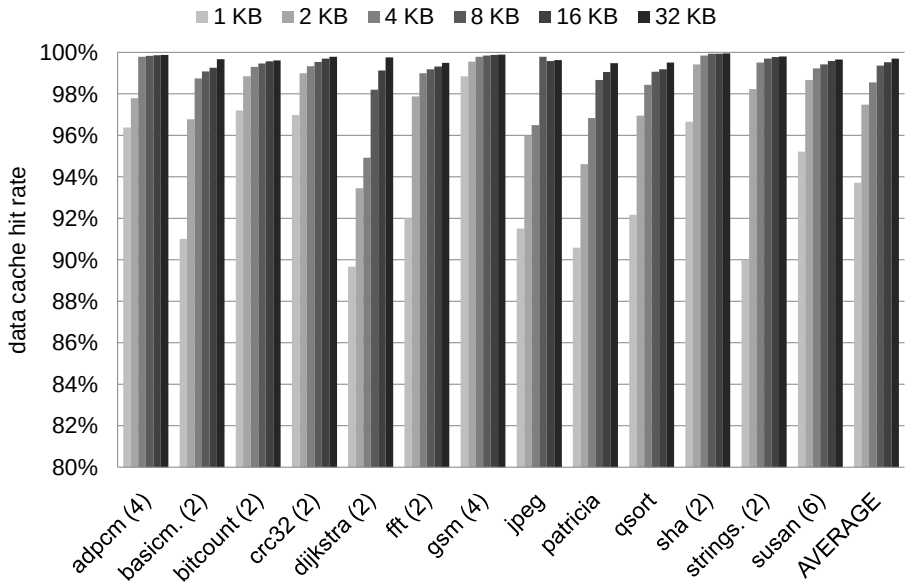

Figure 4.13: Data cache hit rates for the synthetic benchmarks at the `-O2` optimization level.

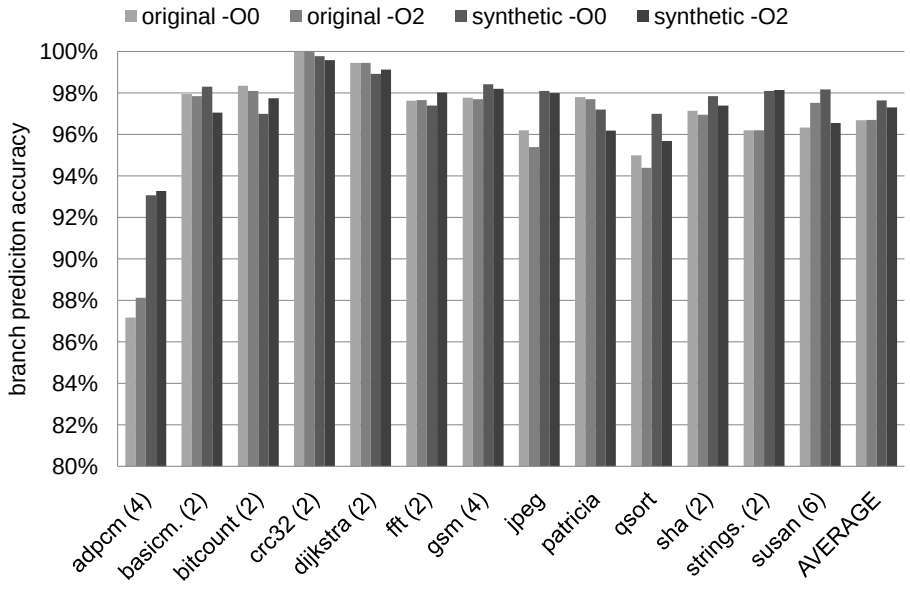

Figure 4.14: Branch prediction rates for the original applications and the synthetic benchmarks.

predictors, i.e., a history predictor is able to predict the branches that are modeled using a modulo operation more accurately.

Detailed cycle-accurate simulation

Figure 4.15 shows CPI for a 2-wide out-of-order processor while varying cache size; these results are obtained through detailed cycle-accurate simulation using PTLSim — the processor configuration is shown in Table 4.5. The synthetics track overall performance across the benchmarks fairly well. For example, `fft` is the benchmark with the highest CPI (due to a large fraction of floating-point instructions) and `sha` the lowest CPI; we observe this for both the real applications and the synthetic benchmarks. We also observe that the synthetic benchmark captures the performance trend as a function of data cache size well, see for example `dijkstra` and `qsort`. The remaining errors come from a number of potential sources. The current data dependency model can be improved to more accurately mimic real application behavior in the synthetic benchmarks (see `bitcount`); also, modeling the branch behavior can be improved upon (see `adpcm`), as well as the data cache behavior (see `stringsearch`). Improving the modeling of

Table 4.5: The baseline processor model considered in our simulations.

Parameter	Configuration
ROB	64 entries
load queue	24 entries
store queue	16 entries
issue queues	2 16-entry issue queues
processor width	2 wide fetch, decode, dispatch, issue, commit
latencies	load (2), mul(3), div(20)
L1 I-cache	32 KB 4-way set-associative, 1 cycle
L1 D-cache	8/16/32 KB 4-way set-associative, 1 cycle
main memory	180 cycle access time
branch predictor	hybrid bimodal/gshare predictor
frontend pipeline	8 stages

these program characteristics is likely to improve the representativeness of the synthetic benchmarks compared to the original applications. Also, different applications may require different R values to faithfully reproduce the application behavior in a synthetic benchmark.

Overall performance across architectures and compilers

Figure 4.16 shows the normalized execution times for the original applications and the synthetic clones across different architectures and compilers and optimization levels; we consider the real machines and a benchmark consolidation setup here and report average numbers. All the results are normalized to the `-O0` optimization level on the 3Ghz Pentium 4 machine. This graph shows that the synthetic benchmarks track the original applications fairly well across architectures and compiler optimization levels. The error in predicting the speedup relative to `-O0` is less than 20% across all machines and optimization levels, with an average error of 7.4%. The synthetics track that the Core i7 yields the best overall performance, and the Itanium 2 the worst. A particularly encouraging result is that the synthetic workload is able to track that the `-O2` and `-O3` optimization levels yield a substantial 25% performance benefit over `-O1` on the Itanium 2 machine but not on the other machines. This performance benefit for the Itanium architecture is due to the fact that Itanium's EPIC architecture is sensitive to compiler optimizations — an EPIC architecture is a statistically scheduled architecture as opposed to a dynamically scheduled out-of-order pro-

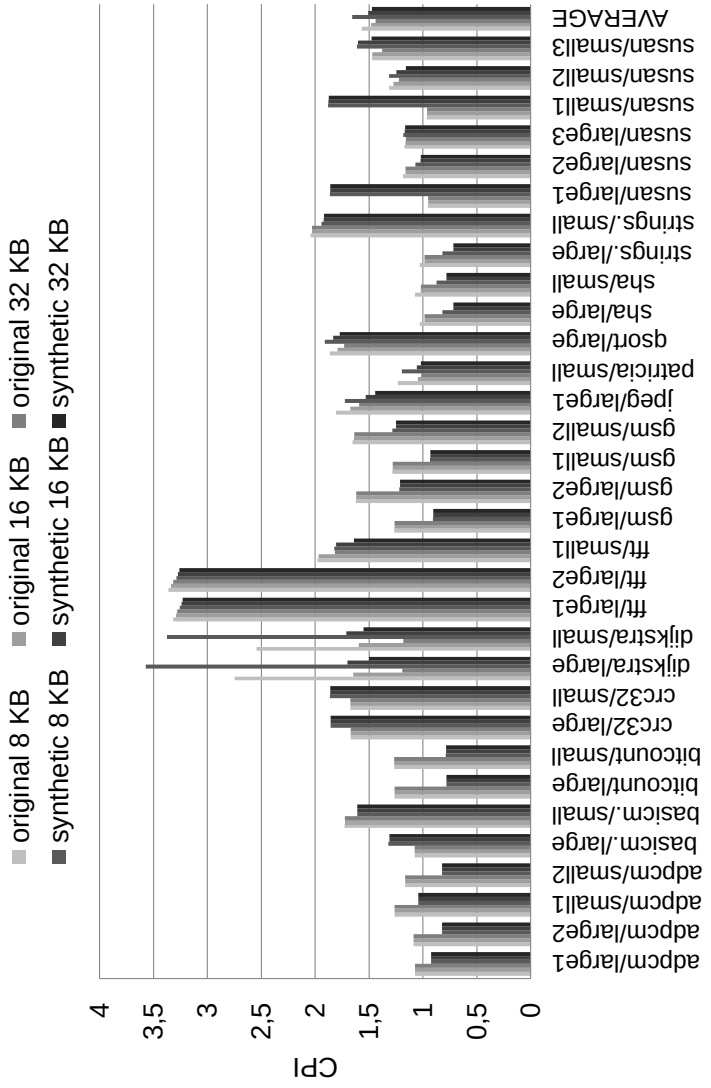

Figure 4.15: CPI for the original and synthetic workloads on a 2-wide out-of-order processor while varying cache size.

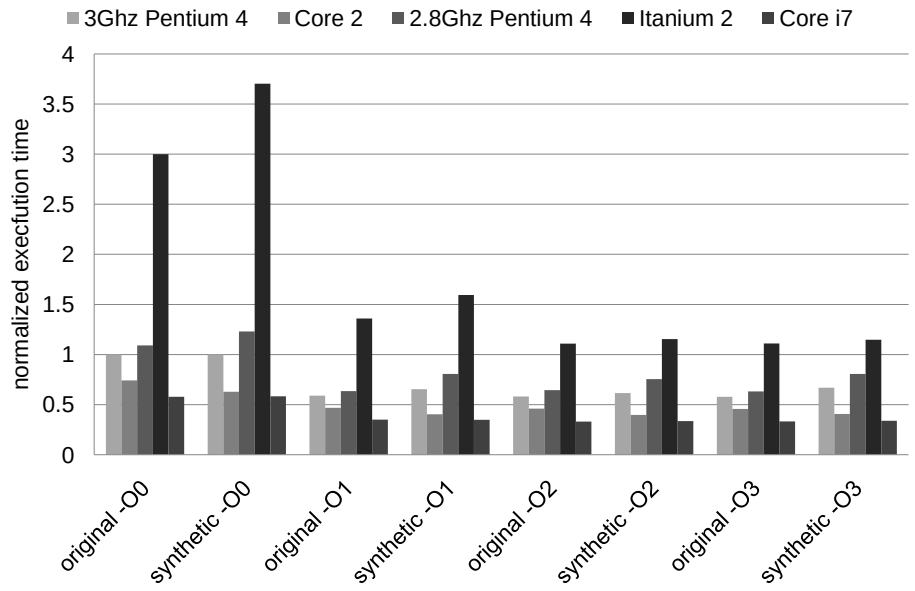

Figure 4.16: Normalized average execution time for the original applications and synthetic benchmarks across architectures and compilers.

cessor, hence compiler optimizations have a more significant impact on overall performance. Clearly, the synthetic benchmarks expose program constructs similar to the real applications that enable the compiler to optimize in a similar vein.

4.5.2 Hiding functional semantics

An important asset of benchmark synthesis is that it hides proprietary information, i.e., it is impossible, or at least very hard, to reverse engineer proprietary information from the synthetic benchmark. One way of evaluating whether this is really achieved is through manual inspection. By comparing the synthetic benchmark against the original application, one can assess whether any proprietary information is still left in the synthetic benchmark. Presumably, companies that plan on using benchmark synthesis will most likely do this validation process very carefully before distributing a synthetic clone.

We now use existing tools for evaluating whether proprietary information is still present in the synthetic benchmark. We therefore use two existing tools, Moss [Aiken, 2003] and JPlag [Malpohl, 1996], which

are used to find plagiarism in software. Moss' main usage has been in detecting plagiarism in programming classes. JPlag is aware of programming language syntax and program structure. The way both tools work is that the user gives two source code files, and the tool returns whether there is any similarity between these two files. When giving the original application and the synthetic benchmark, both Moss and JPlag return that the synthetic benchmark does not provide any similarity with the original application.

4.6 Related work

Our work shares some commonalities with statistical simulation. Statistical simulation [Noonburg and Shen, 1997] [Oskin et al., 2000] [Nussbaum and Smith, 2001] [Eeckhout et al., 2004] [Genbrugge et al., 2006] collects program characteristics from a program execution and subsequently generates a synthetic trace from it which is then simulated on a simple, statistical trace-driven simulator. The important advantage of statistical simulation is that the dynamic instruction count of a synthetic trace is very short, typically a few millions of instructions at most, making it a useful simulation speedup technique for quickly identifying a region of interest in a large microprocessor design space.

Synthetic benchmarks such as Whetstone [Curnow and Wichmann, 1976] and Dhrystone [Weicker, 1984] are manually crafted benchmarks. Manually building benchmarks though is both tedious and time-consuming, and in addition, these benchmarks are quickly outdated. Therefore, recent work proposed automated synthetic benchmark generation [Bell and John, 2005] [Bell et al., 2006] [Joshi et al., 2007] [Joshi et al., 2008a] [Joshi et al., 2008b] [Ganesan et al., 2010] which builds on the statistical simulation approach but generates a synthetic benchmark rather than a synthetic trace.

Our framework borrows some concepts proposed in this related work. Our proposal extends the statistical flow graph [Eeckhout et al., 2004] with loop information. By doing so, our synthetic benchmarks consist of many (nested) loops as observed in real applications. The approach of Bell and John [2005] on the other hand, generates a linear sequence of instructions that is iterated in a big loop until convergence. Our framework also borrows the idea of using the branch transition rate for modeling the branch behavior [Joshi et al., 2008a] and the stride-based memory access pattern modeling approach [Bell and

John, 2005]. The main difference with this prior work though is that (i) we target synthetic benchmarks in a high-level programming language, whereas prior frameworks generated synthetic traces or benchmarks in assembly language, (ii) we generate fine-grained loop structures using the SFGL, and (iii) we use pattern recognition rather than statistics to generate synthetic code sequences.

Code obfuscation [Collberg et al., 1997] converts a program into an equivalent program that is more difficult to understand and reverse engineer. There is a fundamental difference between code obfuscation and benchmark synthesis though. The goal of program obfuscation is to generate a transformed program that is functionally equivalent to the original program, i.e., when given the same input, the transformed program should produce the same output as the original program. The performance characteristics of the transformed program can be very different from the original program. Benchmark synthesis on the other hand generates a synthetic program that exhibits the same performance characteristics as the original program; however, its functionality can be very different. Not having to preserve functionality has an important implication for benchmark synthesis because it allows for generating a synthetic benchmark for a specific input, hence benchmark synthesis can also hide proprietary information as part of the input.

4.7 Summary

The current benchmarking process is typically driven by application benchmarks, i.e., benchmarks that are derived from real-life applications. Although this is an effective approach, there are two major limitations: (i) available benchmarks may not be truly representative of real-life applications (of interest), and (ii) the simulation of contemporary benchmarks is very time-consuming. Code mutation can be used in combination with sampled simulation to combat these limitations; however, the resulting workloads cannot be used for compiler and ISA exploration.

This chapter proposed a novel benchmark synthesis paradigm that generates synthetic benchmarks in a high-level programming language. It generates small but representative benchmarks that can serve as proxies for other applications without revealing proprietary information; and because the benchmarks are generated in a high-level language, they can be used to explore the architecture and compiler

space. The methodology to generate these benchmarks comprises two key steps: (i) profiling a real-world (proprietary) application (that is compiled at a low optimization level) to measure its execution characteristics, and (ii) modeling these characteristics into a synthetic benchmark clone. Our experimental results obtained with our initial framework are promising and demonstrate the feasibility and effectiveness of the approach. We demonstrated good correspondence between the synthetic and original applications across instruction-set architectures, microarchitectures and compiler optimizations; we also pointed out the major sources of error in the benchmark synthesis process. We verified using software plagiarism detection tools that the synthetic benchmark clones indeed hide proprietary information from the original applications.

We also described the potential applications of this benchmark synthesis paradigm: distributing proprietary applications as proxies, drive architecture and compiler research and development, speed up simulation, model emerging and hard-to-setup workloads, and benchmark consolidation.

Chapter 5

Comparing workload design techniques

Your true value depends entirely on what you are compared with.
Bob Wells

The workload generation and reduction techniques discussed in the previous chapters exhibit different strengths and weaknesses. In this chapter, we compare code mutation, sampling, and benchmark synthesis along with the previously proposed input reduction approach. We do this using the following criteria: (i) Do the different techniques yield representative and short-running benchmarks? (ii) Can they be used for both architecture and compiler explorations? (iii) Do they hide proprietary information?

5.1 Introduction

We first briefly revisit the benchmark requirements that we identified in the introduction of this dissertation.

- The benchmarks should reflect their target domain. A benchmark that is not representative of a target domain may lead to a suboptimal design. In addition, given that the processor design cycle takes five to seven years, architects should anticipate future workload characteristics.
- The benchmarks should also be short-running to limit simulation time during design space exploration.

- The benchmarks need to enable both (micro)architecture and compiler research and development, e.g., to evaluate new ISA-extensions.
- The benchmarks must not reveal proprietary information. Industry has the workloads that the user cares about; however, companies are reluctant to release their codes. For this reason, researchers and developers typically have to rely on open-source benchmarks which may not be truly representative for the real-world workloads.

Fulfilling all of the above criteria is non-trivial, and to the best of our knowledge, none of the existing workload generation and reduction techniques address them all.

5.2 Workload design techniques

Before comparing input reduction, code mutation, sampling, and benchmark synthesis against each other, we provide details on the techniques that have not been (fully) described in the previous chapters, namely input reduction, sampling and benchmark synthesis.

5.2.1 Input reduction

The idea of input reduction [KleinOsowski and Lilja, 2002] is to reduce a reference input or to come up with a different input that leads to a shorter running benchmark compared to a reference input while exhibiting similar program behavior. Although the idea of input reduction is simple, doing so in a faithful way is far from trivial.

Most benchmark suites come with a number of inputs. For example, SPEC CPU comes with three inputs. The test input is used to verify whether the benchmark runs properly, and should not be used for performance analysis. The train input is used to guide profile-based optimizations, i.e., the train input is used during profiling after which the system is optimized. The reference input is the one that is to be used for performance measurements. In some cases, researchers and developers also use train inputs to report performance numbers. The primary reason typically is that it takes too long to simulate a benchmark run with a reference input. In particular, simulating a benchmark execution with a

reference input can take multiple weeks to run to completion on today's fastest simulators on today's fastest machines. A train input brings the total simulation time down to a couple hours.

The pitfall using train inputs, and smaller inputs in general, is that they may not be representative of the reference inputs. For example, the working set of a reduced input is typically smaller, hence their cache and memory behavior may stress the memory hierarchy less than the reference input would. Another fundamental problem with this approach is that it takes significant time to come up with the reduced input set for a workload.

KleinOsowski and Lilja [2002] explore the idea of input reduction and they proposed MinneSPEC which collects a number of reduced input sets for some CPU2000 benchmarks. These reduced input sets are derived from the reference inputs using a number of techniques: modifying inputs (for example, reducing the number of iterations), truncating inputs, etc. They propose three reduced inputs: `smred` for short simulations, `mdred` for medium-length simulations, and `lgred` for full-length, reportable simulations. KleinOsowski and Lilja [2002] compare the representativeness of the reduced inputs against the reference inputs by comparing their function-level execution profiles, which appears to be accurate for most benchmarks but not all [Eeckhout et al., 2003].

5.2.2 Sampling

We showed that sampled simulation is very effective at reducing the dynamic instruction count while retaining representativeness and accuracy (see Chapter 3). However, it requires that the simulator is modified to quickly navigate between sampling units and to establish architecture state (register and memory state) and microarchitecture state (content of caches, TLBs, predictors, etc.) at the beginning of the sampling units.

Ringenberg et al. [2005] present intrinsic checkpointing which does not require modifying the simulator. Instead, intrinsic checkpointing rewrites the benchmark's binary and stores the checkpoint (architecture state) in the binary itself. Intrinsic checkpointing provides fix-up checkpointing code consisting of store instructions to put the correct data values in memory and other instructions to put the correct data values in registers [Ringenberg et al., 2005].

The original SimPoint approach [Sherwood et al., 2002] focused on finding representative sampling units based on the basic blocks that are being executed. Follow-on work considered alternative program characteristics such as loops and method calls, which enabled them to identify cross binary sampling units that can be used by architects and compiler builders when studying ISA extensions, and evaluating compiler and software optimizations [Perelman et al., 2007].

5.2.3 Benchmark synthesis

Statistical simulation [Noonburg and Shen, 1997] [Oskin et al., 2000] [Nussbaum and Smith, 2001] [Eeckhout et al., 2004] [Genbrugge et al., 2006] collects program characteristics from a program execution and subsequently generates a synthetic trace from it which is then simulated on a statistical processor simulator. The important advantage of statistical simulation is that the dynamic instruction count of a synthetic trace is very short, typically a few millions of instructions at most. A synthetic trace hides proprietary information very well; however, a synthetic trace cannot be run on real hardware nor on an execution-driven simulator (which is current practice as opposed to trace-driven simulation). Hence, statistical simulation is primarily useful for guiding early-stage design space explorations.

More recent work focused on automated synthetic benchmark generation which builds on the statistical simulation approach but generates a synthetic benchmark rather than a synthetic trace. Although our benchmark synthesis approach shares some commonalities with this prior work, there are important differences as well. For one, in our work we aim at generating synthetic benchmarks in a high-level programming language such as C so that both compiler and architecture developers and researchers can use these benchmarks. Prior work in automated benchmark synthesis generates binaries which limits their usage to architects only, i.e., the synthetic benchmarks cannot be used for compiler research and development. In addition, there are some technical differences as well. For example, whereas prior benchmark synthesis approaches model control flow behavior in a coarse-grain manner, our current work models fine-grained control flow behavior, including (nested) loops, if-then-else structures, etc. Also, we use pattern recognition rather than statistics and distributions for generating synthetic code sequences.

5.3 Comparison

Table 5.1 compares input reduction, code mutation, sampling and benchmark synthesis in terms of a number of dimensions. It is immediately apparent from this table that there is no clear winner. The different techniques represent different trade-offs which makes discussing the differences in more detail interesting and which naturally leads to different use cases for each technique.

Accuracy

Whether the generated workload is representative of the original reference workload is obviously of primary importance. Although it is hard to compare the various workload generation techniques without doing an apples-to-apples comparison — this would require a comparison using the same set of benchmarks and simulation infrastructure — we can make a qualitative statement based on the results in this dissertation.

Sampling is the most accurate approach, followed by code mutation. In Chapter 3 we showed an average performance difference of only 0.29% between full simulation and sampled simulation with NSL-BLRL K=100%, see Figure 3.8. For code mutation, the performance of the mutated binary is also very similar to the original workload: within 1.4% on average (and at most 6%) on real hardware, see Figure 2.17.

Benchmark synthesis has shown medium accuracy; an average performance difference of 7.4% between the synthetic clone and the original workload across a set of compiler optimization levels and hardware platforms, see Figure 4.16. However, the performance difference on a per-benchmark basis can be much higher, as illustrated in Figure 4.15.

The intuitive reason for the higher performance difference of benchmark synthesis is that both sampling and code mutation start from an original application whereas benchmark synthesis identifies a set of program characteristics that when modeled in a synthetic benchmark reflects the performance of the original application — which is non-trivial.

Reduced inputs have shown good accuracy for some benchmarks but very poor accuracy for others. Moreover, compared to sampled simulation, the poor accuracy is not offset by higher simulation speed [Yi et al., 2005].

Simulation time reduction

All techniques except for code mutation aim at reducing the dynamic instruction count so that simulation time is reduced. Simulation time reductions of several orders of magnitude have been reported for sampled simulation (see Chapter 3) and benchmark synthesis (see Chapter 4). Similar simulation time reductions are reported for reduced input data sets [Haskins et al., 2002]. Reducing the dynamic instruction count is not only important for architecture research and development, it is also extremely important in the compiler space, e.g., iterative compilation evaluates a very large number of compiler optimizations in order to find the optimum compiler optimizations for a given program [Cooper et al., 1999] [Kulkarni et al., 2004]. A reduced workload that executes faster will also reduce the overall compiler space exploration time.

Hide proprietary information

Because benchmark synthesis is a bottom-up approach, it succeeds the most in hiding proprietary information, followed by code mutation. On the other hand, code mutation yields more accurate synthetic clones because of its top-down approach. The intrinsic checkpointing approach [Ringenberg et al., 2005] also complicates the understanding of a (proprietary) application; however, representative sampling units will most likely contain valuable information. An important application for these techniques would be to generate synthetic clones for real-world proprietary workloads. This would enable sharing codes among companies in industry. Also, it would be an enabler for industry to share their applications with their research partners in academia without revealing proprietary information.

Architecture and compiler exploration

All techniques can be used to drive microarchitecture research and development; however, only a few can be used for compiler and ISA exploration. The reason is that techniques, such as code mutation, sampling, statistical simulation and benchmark synthesis at the binary level, operate on binaries and not on source code which eliminates their usage for compiler and ISA exploration. On the other hand, sampling that identifies representative loops and function calls can be used to

	Input reduction	Code mutation	Sampling	Benchmark synthesis at binary level	Benchmark synthesis at HLL level
Accuracy wrt reference workloads	Medium-Poor	Medium-High	High	Medium	Medium
Reduces simulation time	Yes	No	Yes	Yes	Yes
Hides proprietary information	No	Yes	No	Yes	Yes
Enables microarchitecture exploration	Yes	Yes	Yes	Yes	Yes
Enables compiler & ISA exploration	Yes	No	Partially	No	Yes
Can model emerging workloads	No	No	No	Yes	Yes

Table 5.1: Comparison of workload generation and reduction techniques [Van Ertvelde and Eeckhout, 2010b].

drive compiler research, as does benchmark synthesis at the HLL level.

Model emerging workloads

Benchmark synthesis can also be used to generate emerging and future workloads. In particular, one can generate a workload profile with performance characteristics that are to be expected for future emerging workloads. For example, one could generate a synthetic workload with very large working sets, or random memory access patterns, or complex control flow behavior. The synthetic benchmarks generated from these profiles can then be used to explore design alternatives for future computer systems.

Multi-threaded workloads

Contemporary computer systems all feature multicore processors, which obviously has its repercussions on benchmarking for both hardware and software. Recent work in workload generation and reduction has focused almost exclusively on single-threaded workloads, except for a few studies in sampling (see for example [Wenisch et al., 2006b]) and benchmark synthesis (see for example [Hughes and Li, 2008]).

5.4 Summary

In this chapter, we compared recently proposed workload generation and reduction techniques, and we came to the conclusions that there is no clear winner, i.e., the different techniques represent different trade-offs. The trade-off between code mutation and benchmark synthesis is that synthetic benchmarks may be less accurate and representative with respect to the real applications compared to mutated binaries; however, benchmark synthesis hides proprietary information more adequately and it yields short-running benchmarks.

Chapter 6

Conclusion

We can chart our future clearly and wisely only when we know the path which has led to the present.
Adlai Stevenson

This dissertation investigated several challenges when using benchmarks in computer architecture research and development. In this chapter, we first summarize these challenges and then detail the conclusions that can be drawn from this research work. In addition, we highlight interesting research topics that could be investigated further in the future, with special emphasis on code mutation and benchmark synthesis.

6.1 Summary

The growing complexity of contemporary microarchitectures necessitates the use of benchmark programs in computer science and engineering research and development, i.e., computer architects and compiler designers use benchmarks to evaluate their products and research ideas. Consequently, several organizations such as SPEC, TPC, EEMBC, etc., have released standard application benchmark suites to streamline this performance evaluation process. Nevertheless, computer architects and engineers still face several important benchmarking challenges.

For one, industry vendors hesitate to disseminate proprietary applications to academia and third-party vendors. By consequence, the benchmarking process is typically driven by standardized, open-source

benchmarks which may be very different from and likely not representative of the real-world applications of interest. In addition, available benchmark suites are often outdated because the application space is constantly evolving and developing new benchmark suites is extremely time-consuming (and costly).

Second, contemporary application benchmark suites like SPEC CPU2006 execute trillions of instructions in order to stress contemporary and future processors in a meaningful way. This has significant implications for simulation-based design space exploration, i.e., it is infeasible to simulate entire application benchmarks using detailed cycle-accurate simulators. Simulating only one second of real time may lead to multiple hours or days of simulation time, even on today's fastest simulators running on today's fastest machines.

Finally, coming up with a benchmark that is representative, short-running yet versatile is another major challenge. A benchmark should enable both (micro)architecture and compiler research and development. Although existing benchmarks satisfy this requirement, this is typically not the case for workload reduction techniques that reduce the dynamic instruction count in order to address the simulation challenge. These techniques often operate on binaries and not on source code which eliminates their utility for compiler exploration and ISA exploration.

In the following subsections, we briefly highlight the major findings and contributions of this dissertation to the workload generation for microprocessor performance evaluation.

6.1.1 Code mutation

Code mutation is a novel methodology for constructing benchmarks that hide the functional semantics of proprietary applications while exhibiting similar performance characteristics. The benchmark mutants then can serve as proxies for the proprietary applications during benchmarking experiments. The code mutation framework derives benchmarks from proprietary applications by exploiting two key observations: (i) miss events have a dominant impact on performance on contemporary microprocessors, and (ii) many variables of contemporary applications exhibit invariant behavior at run-time. The novelty of our idea is to approximate application performance characteristics by retaining memory access and control flow behavior while mutating the

remaining application code. We therefore compute program slices for memory access and/or control flow operations trimmed through value and branch profiles, and subsequently mutate the instructions not appearing in these slices. The end result is a benchmark mutant that can be shared among third-party industry vendors, as well as between industry and academia. This could make the benchmarking process in industry both more accurate and more straightforward, and the performance evaluation process in academia more realistic.

We explored a number of approaches to code mutation — these approaches differ in the way they preserve the proprietary application's memory access and control flow behavior in the mutant. We found CFO plus ECF (Control Flow Operation slicing and Enforced Control Flow) the approach that represents the best trade-off between accuracy and information hiding. This approach computes control flow slices for frequently executed, non-constant branches, and mutates instructions that are not part of any of these slices. The slices are trimmed using constant value profiles to make more instructions eligible for code mutation. For this approach, code mutation mutates up to 90% of the binary, up to 50% of the dynamically executed instructions, and up to 35% of the at-run-time-exposed inter-operation data dependencies. We also demonstrated that the performance characteristics of the mutants correspond well with those of the original applications; for CFO plus ECF, the average execution time deviation on hardware is 1.4%.

6.1.2 Cache state warmup for sampled simulation through NSL-BLRL

Architectural simulation is an essential tool for microarchitectural research to obtain insight into the cycle-level behavior of current microprocessors. However, architectural simulations are extremely time-consuming, especially if contemporary benchmarks suites need to be simulated to completion. Sampled simulation is a well-known solution for speeding up architectural simulation; the key idea is to only simulate a small sample from a complete benchmark execution in a cycle-accurate manner. An important problem in sampled simulation is to warmup the microarchitectural state at the beginning of a sampling unit.

We proposed a hybrid cache state warmup approach that combines cache state checkpointing through NSL (No-State-Loss) with BLRL

(Boundary Line Reuse Latency) into NSL-BLRL. The key idea is to truncate the NSL stream of memory references in a pre-sampling unit using BLRL information; the truncated NSL stream then serves as a cache state checkpoint.

We demonstrated that this approach yields several benefits over prior work: it yields substantial simulation time speedups compared to BLRL (up to 1.4× under fast-forwarding and up to 14.9× under checkpointing) and significant reductions in disk space requirements compared to NSL (30% on average). Also, NSL-BLRL is more broadly applicable than the MHS and TurboSMARTS approach because the NSL-BLRL warmup info is independent of the cache block size.

6.1.3 High-level language benchmark synthesis

We presented a novel benchmark synthesis framework for generating synthetic benchmarks that are small though representative for other applications. The framework comprises two key steps: (i) profiling a real-world (proprietary) application (that is compiled at a low optimization level) to measure its execution characteristics, and (ii) modeling these characteristics into a synthetic benchmark clone. The key novelty is that the synthetic benchmarks are generated in a high-level programming language to enable both architecture and compiler research — prior work in benchmark synthesis generated synthetic benchmarks at the binary level.

Furthermore, we introduced a novel structure to capture a program's control flow behavior in a statistical way. This structure enables our framework to generate conditional control flow behavior, (nested) loops and function calls in the synthetic benchmark — prior work instead generated a linear sequence of basic blocks. We demonstrated that modeling this behavior is necessary to show good correspondence between the synthetic and original applications across instruction-set architectures, microarchitectures and compiler optimizations.

We also elaborated on the potential applications of this benchmark synthesis paradigm: distributing proprietary applications as proxies, drive architecture and compiler research and development, speed up simulation, model emerging and hard-to-setup workloads, and benchmark consolidation.

6.2 Future work

If we examine our thoughts,
we shall find them always occupied with the past and the future.
Blaise Pascal

The past is but the past of a beginning.
H. G. Wells

Because both code mutation and high-level language benchmark synthesis are novel methodologies, there are several research opportunities that can be investigated in the future. These opportunities lie mainly in improving the accuracy, efficacy and applicability of both approaches. In the remaining of this chapter, we detail this possible future work.

6.2.1 Code mutation

A first interesting area of research could be to further improve the information hiding aspect of our code mutation methodology. The current number of instructions that can be mutated may still be insufficient to persuade companies to distribute their proprietary applications as benchmark mutants. To further remove and/or hide the intellectual property of a proprietary application, we can employ novel and more aggressive program analyses and transformations. More in particular, we can make the following suggestions:

- A study by Calder et al. [1997] shows that many variables exhibit semi-invariant behavior. A semi-invariant variable is one that cannot be identified as a constant at compile time, but has a high degree of invariant behavior at run time. One potential research direction could be to exploit this semi-invariant program behavior in order to mutate an even larger fraction of the proprietary application. Currently, program slices are trimmed using invariant program behavior only. An improvement could be to also use semi-invariant behavior to reduce the size of these slices even further. This could be achieved by recording the different values of the semi-invariant variables, and then replay these values in the mutant at run time. By doing so, more instructions

become eligible for code mutation.

- Another research direction could be to deploy existing obfuscation techniques. Although the obfuscation techniques that we evaluated had a significant impact on the performance behavior of an application, other code transformations may be more appropriate for code mutation — Collberg et al. [1997] evaluate a number of obfuscating transformations in terms of how much overhead they add to the obfuscated application. It may also be possible to strengthen existing obfuscation techniques: (i) by exploiting the fact that code transformations are allowed to change the functional behavior of a program, and (ii) by specializing obfuscation techniques for a particular input.

- Yet another research direction is to study the feasibility of intermingling two or more benchmark mutants together into one single mutant. A single mutant would enable us to make reverse engineering even more difficult, e.g., by introducing artificial dependencies between two or more merged mutants. Furthermore, a single mutant could represent an entire suite of applications in one individual package.

- Finally, because input data sets are often proprietary, it might be interesting to examine if code mutation can be applied to input data sets as well, i.e., by mutating the parts of the input that do not affect control flow behavior.

Another possible research area is to improve the applicability of our code mutation methodology. We identify the following key research directions along this line:

- Extending the code mutation concept to multi-threaded workloads as well as applications written in type-safe managed languages such as Java and C#. We believe that both are possible — the general concept of code mutation applies to these workloads as well while posing a number of additional constraints. In particular, multi-threaded workloads incur an additional constraint in that accesses to shared memory should be preserved in the mutated benchmark in order to exhibit similar inter-thread communication in the mutant as in the original application. Hence, slices will need to be computed for shared-memory accesses, and

instructions appearing in these slices should not be overwritten through code mutation. For type-safe managed languages, code mutation will be restricted by type safety, i.e., an operation can only be overwritten by another operation if both operate on the same type.

- Our current framework mutates the proprietary application at the binary level. An alternative approach would mutate the application at an intermediate code level or even at the source code level, so that the mutant can be compiled and optimized for a particular ISA of interest.
- It may be possible to combine code mutation with intrinsic checkpoints in order to generate mutants that are short running. Intrinsic checkpointing [Ringenberg et al., 2005] decreases simulation time by augmenting binaries to contain checkpointing instructions that allow for the rapid execution of important portions of code. The intrinsic checkpointing instructions recreate the architecture state prior to such a code interval. Hence, instructions between two code intervals do not need to be simulated.

Finally, yet another interesting research topic is to further quantify the effectiveness of code mutation, i.e., quantify to what extent code mutation is able to make reverse engineering — in the form of disassembly followed by decompilation — more difficult to be performed. One possibility is to map our proposed metrics for quantifying the effectiveness of code mutation onto a couple of threat models. However, the difficulty here is that there is always a 'human factor' involved.

6.2.2 High-level language benchmark synthesis

The initial results of our benchmark synthesis framework are promising and therefore we believe there are many opportunities for future work. A first interesting area of research could be to improve the accuracy of our methodology. The key challenge is to model (most of) the program characteristics that impact a program's performance, but do this without revealing proprietary information — there is a trade-off in representativeness versus hiding proprietary information. More in particular, the following aspects could be modeled more accurately:

- We use a stride-based memory access model that is microarchitecture dependent and does not model memory-level parallelism.

Although this model accurately mimics the overall miss rate of the original application, it may not resemble the performance of the original application closely. Future work could investigate more complicated access behavior models. Ganesan et al. [2010] investigate a stride-based memory access model that incorporates memory-level parallelism information. An additional challenge is to model data dependencies following dependence patterns seen in the original workload. This can be done by capturing the number of instructions between the production and the consumption of a data value (at the source level).

- If the goal is to generate shorter workloads only, without worrying about intellectual property, we could borrow techniques from reverse-engineering to enhance our pattern matcher. This may translate in more accurate synthetic benchmarks.

In order to improve the applicability of our benchmark synthesis methodology, we present the following research directions:

- An important research direction is to extend our framework towards multithreaded workloads. Hughes and Li [2008] propose to construct synchronized statistical flow graphs that incorporate inter-thread synchronization and sharing behavior to capture the complex characteristics and interactions among multiple threads. It is worth investigating whether similar techniques can be used in order to synthesize multithreaded workloads at a high-level programming language.

- Subsequently, it would be interesting to extend our framework towards more complex workloads, i.e., commercial workloads such as databases and web servers. Commercial workloads typically exhibit more I/O behavior, leading to a potential performance bottleneck. The key challenge here is thus to model these potential performance bottlenecks. Modeling this behavior can be done using existing application programming interfaces — this is an advantage of generating synthetic benchmarks at a high level programming language. This will enable our framework to be used in the high-end server market segment as well.

A final research topic concerns the determination of an application's optimal reduction factor, i.e., different applications probably re-

quire different reduction factors to reproduce the behavior in a synthetic benchmark. If the reduction factor is underestimated, we sacrifice simulation speed. If the reduction factor is overestimated, simulation results will be inaccurate. One possibility is to tune the reduction factor so that the major program phases are still preserved in the synthetic benchmark.

Bibliography

[Agarwal et al., 2000] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 248–259, 2000.

[Agrawal and Horgan, 1990] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 246–256, 1990.

[Aho et al., 2006] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[Aiken, 2003] A. Aiken. Moss: A system for detecting software plagiarism, `http://theory.stanford.edu/~aiken/moss`, 2003.

[Auslander et al., 1996] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 149–159, 1996.

[Austin et al., 2002] T. M. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[Bader et al., 2005] D. 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 IEEE International Symposium on Workload Characterization (IISWC)*, pages 163–173, 2005.

[Barr et al., 2005] K. C. Barr, H. Pan, M. Zhang, and K. Asanovic. Accelerating multiprocessor simulation with a memory timestamp record. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 66–77, 2005.

[Bell et al., 2006] R. H. Bell, R. R. Bhatia, L. K. John, J. Stuecheli, J. Griswell, P. Tu, L. Capps, A. Blanchard, and R. Thai. Automatic testcase synthesis and performance model validation for high performance PowerPC processors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 154–165, 2006.

[Bell and John, 2005] R. H. Bell and L. K. John. Improved automatic testcase synthesis for performance model validation. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 111–120, 2005.

[Black and Shen, 1998] B. Black and J. P. Shen. Calibration of microprocessor performance models. *IEEE Computer*, 31(5):59–65, 1998.

[Blackburn et al., 2006] 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. B. 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 annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

[Calder et al., 1997] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 259–269, 1997.

[Calder et al., 1999] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction-Level Parallelism*, 1, 1999.

[Cao Minh et al., 2008] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multiprocessing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 35–46, 2008.

[Chiou et al., 2007] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. H. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. FPGA-accelerated

simulation technologies (fast): Fast, full-system, cycle-accurate simulators. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 249–261, 2007.

[Collberg et al., 1997] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, 1997, `http://www.cs.auckland.ac.nz/~collberg/`.

[Conte et al., 1998] T. M. Conte, M. A. Hirsch, and W. mei W. Hwu. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Transactions on Computers*, 47(6):714–720, 1998.

[Conte et al., 1996] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 468–477, 1996.

[Cooper et al., 1999] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, 1999.

[Curnow and Wichmann, 1976] H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, 1976.

[De Bus et al., 2003] B. De Bus, D. Kästner, D. Chanet, L. Van Put, and B. De Sutter. Post-pass compaction techniques. *Communications of the ACM*, 46(8):41–46, 2003.

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

[Eeckhout et al., 2004] L. Eeckhout, R. H. Bell, B. Stougie, K. De Bosschere, and L. K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 350–363, 2004.

[Eeckhout et al., 2005] L. Eeckhout, Y. Luo, K. De Bosschere, and L. K. John. BLRL: Accurate and efficient warmup for sampled processor simulation. *The Computer Journal*, 48(4):451–459, 2005.

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

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

[Eyerman et al., 2008] S. Eyerman, L. Eeckhout, and J. E. Smith. Studying compiler optimizations on superscalar processors through interval analysis. In *Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, volume 4917, pages 114–129, 2008.

[Ganesan et al., 2010] K. Ganesan, J. Jo, and L. K. John. Synthesizing memory-level parallelism aware miniature clones for SPEC CPU2006 and ImplantBench workloads. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 33–44, 2010.

[Ge et al., 2005] J. Ge, S. Chaudhuri, and A. Tyagi. Control flow based obfuscation. In *Proceedings of the ACM Workshop on Digital Rights Management (DRM)*, pages 83–92, 2005.

[Genbrugge et al., 2006] D. Genbrugge, L. Eeckhout, and K. De Bosschere. Accurate memory data flow modeling in statistical simulation. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 87–96. ACM, 2006.

[Gupta et al., 2005] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 263–272, 2005.

[Guthaus et al., 2001] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization (WWC)*, pages 3–14, 2001.

[Haskins et al., 2002] Haskins, K. Skadron, A. KleinOsowski, and D. J. Lilja. Techniques for accurate, accelerated processor simulation: Analysis of reduced inputs and sampling. Technical report, Charlottesville, VA, USA, 2002.

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

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

[Hennessy and Patterson, 2003] J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, third edition, 2003.

[Henning, 2000] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 2000.

[Henning, 2006] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

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

[Hughes and Li, 2008] C. Hughes and T. Li. Accelerating multi-core processor design space evaluation using automatic multi-threaded workload synthesis. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 163–172, 2008.

[Iyengar et al., 1996] V. S. Iyengar, L. Trevillyan, and P. Bose. Representative traces for processor models with infinite cache. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 62–72, 1996.

[Joshi et al., 2006a] 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, 2006.

[Joshi et al., 2006b] A. M. Joshi, L. Eeckhout, R. H. Bell, and L. K. John. Performance cloning: A technique for disseminating proprietary applications as benchmarks. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 105–115, 2006.

[Joshi et al., 2008a] A. M. Joshi, L. Eeckhout, R. H. Bell., and L. K. John. Distilling the essence of proprietary workloads into miniature benchmarks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(2), 2008.

[Joshi et al., 2007] A. M. Joshi, L. Eeckhout, and L. K. John. Exploring the application behavior space using parameterized synthetic benchmarks. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, page 412, 2007.

[Joshi et al., 2008b] A. M. Joshi, L. Eeckhout, L. K. John, and C. Isen. Automated microprocessor stressmark generation. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 229–239, 2008.

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

[KleinOsowski and Lilja, 2002] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, 2002.

[Kulkarni et al., 2004] P. Kulkarni, S. Hines, J. Hiser, D. B. Whalley, J. W. Davidson, and D. L. Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 171–182, 2004.

[Laha et al., 1988] S. Laha, J. H. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11):1325–1336, 1988.

[Lee et al., 1997] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 330–335, 1997.

[Lipasti and Shen, 1996] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the Annual*

ACM/IEEE International Symposium on Microarchitecture (MICRO), pages 226–237, 1996.

[Lipasti et al., 1996] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 138–147, 1996.

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

[Luo et al., 2005] Y. Luo, L. K. John, and L. Eeckhout. Sma: A self-monitored adaptive cache warm-up scheme for microprocessor simulation. *International Journal of Parallel Programming*, 33(5):561–581, 2005.

[Malpohl, 1996] G. Malpohl. JPlag: Detecting software plagiarism, `https://www.ipd.uni-karlsruhe.de/jplag`, 1996.

[Martin et al., 2005] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[Maynard et al., 1994] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 145–156, 1994.

[McMahon, 1986] F. H. McMahon. Livermore FORTRAN kernels: A computer test of the numerical performance range. Technical report, Lawrence Livermore National Laboratories, Livermore, California, 1986.

[Moore, 1998] G. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[Mukherjee et al., 2002] S. S. Mukherjee, S. V. Adve, T. Austin, J. Emer, and P. S. Magnusson. Performance simulation tools. *IEEE Computer*, 35:38–39, 2002.

[Narayanasamy et al., 2006] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 216–227, 2006.

[Noonburg and Shen, 1997] D. B. Noonburg and J. P. Shen. A framework for statistical modeling of superscalar processor performance. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 298–309, 1997.

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

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

[Oskin et al., 2000] M. Oskin, F. T. Chong, and M. K. Farrens. HLS: combining statistical and symbolic simulation to guide microprocessor design. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 71–82, 2000.

[Perelman et al., 2007] E. Perelman, J. Lau, H. Patil, A. Jaleel, G. Hamerly, and B. Calder. Cross binary simulation points. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 179–189, 2007.

[Phansalkar et al., 2004] A. Phansalkar, A. Joshi, L. Eeckhout, and L. John. Four generations of SPEC CPU benchmarks: what has changed and what has not. Technical report, ECE, The University of Texas at Austin, 2004.

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

[Pöss and Floyd, 2000] M. Pöss and C. Floyd. New TPC benchmarks for decision support and web commerce. *SIGMOD Record*, 29(4):64–71, 2000.

[Ringenberg et al., 2005] J. Ringenberg, C. Pelosi, D. W. Oehmke, and T. N. Mudge. Intrinsic checkpointing: A methodology for decreasing simulation time through binary modification. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 78–88, 2005.

[Shao et al., 2005] M. Shao, A. Ailamaki, and B. Falsafi. DBmbench: fast and accurate database workload representation on modern microarchitecture. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 254–267, 2005.

[Sherwood et al., 2002] 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, 2002.

[Skadron et al., 2003] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai. Challenges in computer architecture evaluation. *IEEE Computer*, 36(8):30–36, 2003.

[Smith and Sohi, 1995] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83:1609–1624, 1995.

[Sreenivasan and Kleinman, 1974] K. Sreenivasan and A. J. Kleinman. On the construction of a representative synthetic workload. *Communications of the ACM*, 17(3):127–133, 1974.

[Srivastava and Eustace, 1994] A. Srivastava and A. Eustace. ATOM - a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, 1994.

[Tip, 1995] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[Van Biesbrouck et al., 2005] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation.

In *Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, pages 47–67, 2005.

[Van Ertvelde and Eeckhout, 2008] L. Van Ertvelde and L. Eeckhout. Dispersing proprietary applications as benchmarks through code mutation. In *The International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 201–210, 2008.

[Van Ertvelde and Eeckhout, 2010a] L. Van Ertvelde and L. Eeckhout. Benchmark synthesis for architecture and compiler exploration. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 106–116, 2010.

[Van Ertvelde and Eeckhout, 2010b] L. Van Ertvelde and L. Eeckhout. Workload reduction and generation techniques. *IEEE Micro*, 30(6), Nov/Dec 2010.

[Van Ertvelde et al., 2008] L. Van Ertvelde, F. Hellebaut, and L. Eeckhout. Accurate and efficient cache warmup for sampled processor simulation through NSL-BLRL. *The Computer Journal*, 51(2):192–206, 2008.

[Van Ertvelde et al., 2006] L. Van Ertvelde, F. Hellebaut, L. Eeckhout, and K. De Bosschere. NSL-BLRL: Efficient cache warmup for sampled processor simulation. In *Proceedings of the Annual Simulation Symposium (ANSS)*, pages 168–177, 2006.

[Verplaetse et al., 2000] P. Verplaetse, J. Van Campenhout, and D. Stroobandt. On synthetic benchmark generation methods. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, pages 213–216, 2000.

[Weicker, 1984] R. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[Weiser, 1981] M. Weiser. Program slicing. *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, pages 439–449, 1981.

[Weiser, 1982] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

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

[Wenisch et al., 2006b] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. Simflex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006.

[Williams, 1977] J. N. Williams. The construction and use of a general purpose synthetic program for an interactive benchmark on demand paged systems. In *Communications of the ACM*, pages 459–465, 1977.

[Wong and Morris, 1988] W. S. Wong and R. J. T. Morris. Benchmark synthesis using the LRU cache hit function. *IEEE Transactions on Computers*, 37(6):637–645, 1988.

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

[Yi et al., 2005] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 266–277, 2005.

[Yi et al., 2006] J. J. Yi, H. Vandierendonck, L. Eeckhout, and D. J. Lilja. The exigency of benchmark and compiler drift: Designing tomorrows processors with yesterdays tools. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 75–86, 2006.

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

[Zhang et al., 2005] X. Zhang, R. Gupta, and Y. Zhang. Cost and precision tradeoffs of dynamic data slicing algorithms. *ACM Transactions on Programming Language Systems (TOPLAS)*, 27(4):631–661, 2005.

[Zilles and Sohi, 2000] C. B. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. In *Proceed-*

ings of the Annual International Symposium on Computer Architecture (ISCA), pages 172–181, 2000.