UNIVERSITY OF CALIFORNIA,
IRVINE

# Just-in-Time Compilation Techniques for Hardware/Software Co-Designed Processors

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

*Marcelo Cintra*

2015

# DEDICATION

*To my beloved wife, Monique, and my beloved children,*
*Sofia, Lucas Anthony and Benjamin.*

*"Love, just like knowledge, is a treasure that the more we spread,*
*the more we have."*
*Joanna de Ângelis*

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF LISTINGS

# ACKNOWLEDGMENTS

I am deeply indebted to many people who made this work possible.

First and foremost, I thank my advisor Professor Michael Franz, for giving me the opportunity to work with him. My doctoral endeavor with Michael started eight years ago, with an email that I sent him with the title: Hogwarts - Prospective Graduate Student, as requested on his webpage. After a few weeks, I was so glad to receive his reply saying that he would be very happy to work with me, and would recommend my acceptance. Due to some problems in my online application, I actually first received a letter of denial from UCI. From that moment, I saw Michael's dedication in assisting his students to resolve and overcome any challenges or difficulties, which continued after I started my studies. Without his continued assistance, not only on this matter, but also in many others along the journey, this work would not have come to completion. I thank Michael for providing such a great research environment for me, for his encouragement and critical feedback that did so much to improve and shape this work, and above all, for sharing his knowledge and experiences in our countless meetings and almost daily lunches.

I thank the members of my thesis committee, Profs. Michael Franz (Chair), Harry Xu, and Ian Harris, for taking the time to serve on my Advancement and Final Defense committees. Their critical feedback, questions, and comments have helped to shape and improve the quality of this work. I also thank the Brazilian National Council for Scientific and Technological Development, for my research fellowship, award #200973/2007-0, which supported my first two years of study at UCI.

I give special thanks to my dear friend Dr. Guilherme Ottoni, for giving me the opportunity to work with him at Intel, for mentoring my initial steps on Dynamic Binary Translation(DBT) research, which later allowed me to identify new research opportunities to leverage my JIT compilation expertise. I also thank Dr. Edson Borin and Dr. David "Pardo" Keppel, for the countless technical conversations that helped me to: 1) understand the internals of the Efficeon Code Morphing Software, and 2) to make the best design decisions to implement the techniques of this dissertation.

I also thank the entire staff of the Donald Bren School of Information and Computer Science, and the Department of Electrical Engineering and Computer Science of UCI, from whom I learned a lot from the courses and seminars that I have taken, which gave me the knowledge necessary for solving the challenging technical problems of my research. I thank Kris Bolcer and Gina Anzivino, from the Office of Student Affairs at the Donald Bren School of Information and Computer Science, for their precious advice and assistance in the beginning of my PhD endeavor. I especially thank Karina Bocanegra for her advice and effectiveness in dealing with all the bureaucracy in the later part of my academic career, which allowed me to focus on my studies.

I thank my colleagues at the Programming Systems Lab (PSL) at Intel. In particular, I am very grateful for having had the opportunity to work with Cheng Wang and Youfeng Wu, who greatly contributed to make the best design choices during the implementation of our infrastructure for bytecode acceleration. I thank Cheng with whom I worked closely on much of the implementation work, and for his insightful comments and his efficient solution to the problem of inferfacing our acceleration framework with the underlying co-designed processor.

I would like to express my gratitude towards my fellow graduate students at the Secure Systems and Software Laboratory that have made graduate school a wonderful experience, and from whom I've learned a great deal. In particular, I would like to thank Gregor Wagner, with whom I shared office, Michael Bebenita, for the opportunity to collaborate on his research on trace compilation, and Mason Chang, for this help in improving my presentation skills, and for making me laugh a lot with his jokes. I am also grateful to the post-doctoral researchers Dr. Andreas Gal, Dr. Christian Wimmer, Dr. Stefan Brunthaler, and Dr. Per Larsen for their assistance guiding my research work. In special, I would like to thank Dr. Stefan Brunthaler for his feedback on my TACO journal paper, and Dr. Per Larsen, for his constructive critical feedback on this dissertation, which made it easier to understand.

I thank all the masters, teachers, mentors, and inspiring academic research exemplars I had in my entire school time in Brazil. In special: Professors Gersinei, Creuza, Bosco, Eduardo, Padre Tabir, Dr. Edson Caceres, Dr. Marcelo Siqueira, Dr. Ronaldo Ferreira, Dr. Fabio Martinez, and Dr. Nahri Moreano. I give warm thanks to my Master's advisor at UNICAMP, Dr. Guido Araujo, for opening the doors for my international career, and for guiding and mentoring me in my graduate studies.

I would also like to thank my friends in the Bay area, who I could not possibly list all in here, who helped me to relax and to recover my energy. My special thank goes to my dear friends Samuel Goto and Daniela Goto for their support and incentive in the final phase of my studies, and for letting me use their quiet house to write a significant portion of my dissertation while they were on vacation in Brazil. I also thank Samuel for proofreading portions of my dissertation.

I am most grateful to my parents, Jair Cintra and Maria Cintra, and my brothers, Jean Cintra and Joao Paulo Cintra, for their love shown in the form of the encouragement, support which gave me the strength to overcome many of the obstacles during my studies.

Finally, and most importantly, I thank my wife Monique. First, for giving up our financially comfortable life in Brazil and joining me on this journey. My words are not enough to express my feelings of gratitude for her love, care, encouragement, and understanding, which has always been a source of strength and support in my life. I am eternally in debt for her self-abnegation for the benefit of our family, and for encouraging me to continue my PhD studies in the moments I felt weak, especially when I was unable to help her as much as I wanted, and should.

I thank God, for the joy of life, and for all the opportunities to learn, and to contribute towards a better and peaceful society.

# CURRICULUM VITAE

## Marcelo Cintra

### EDUCATION

**Doctor of Philosophy in Computer Science** **2015**
University of California, Irvine *Irvine, California*

**Master of Science in Computer Science** **2000**
University of Campinas (UNICAMP) *Campinas, SP, Brazil*

**Bachelor of Science in Computer Science** **1998**
Federal University of Mato Grosso do Sul (UFMS) *Campo Grande, MS, Brazil*

### RESEARCH EXPERIENCE

**Research Scientist** **2010-Present**
Programming Systems Lab - Intel Labs *Santa Clara, CA*

**Graduate Research Assistant** **2008**
University of California, Irvine *Irvine, California*

### TEACHING EXPERIENCE

**Computer Engineering Lecturer** **2006–2007**
Catholic University Dom Bosco (UCDB) *Campo Grande-MS, Brazil*
Courses taught: Analysis of Algorithms, Compilers, Computer Architetures.

**Computer Science Lecturer** **2006–2007**
University Center of Campo Grande (UNAES) *Campo Grande-MS, Brazil*
Courses taught: Analysis of Algorithms, Compilers, Java Programming.

**Computer Science Lecturer** **2004–2005**
Catholic University of Brasilia (UCB) *Brasilia-DF, Brazil*
Courses taught: Analysis of Algorithms, Compilers.

**Information Systems Lecturer** **2004–2005**
University Center of Brasilia (UNIEURO) *Brasilia-DF, Brazil*
Courses taught: Operating Systems, Data Structures and Java Programming.

**Information Systems Lecturer** **2002–2003**
Faculdade Noroeste de Minas (FINOM) *Paracatu-MG, Brazil*
Courses taught: Algorithms, Data Structures.

## SELECTED HONORS AND AWARDS

**CNPq Graduate Research Fellowship** **2008–2010**
The Brazilian National Council for Scientific and Technological Development

**FAPESP Graduate Research Fellowship** **1998–2000**
Sao Paulo Research Foundation

**CNPq Junior Scientific Initiation Fellowship** **1996–1997**
The Brazilian National Council for Scientific and Technological Development

## REFEREED JOURNAL PUBLICATIONS

**Global array reference allocation** **2002**
ACM Transactions on Design Automation of Electronic Systems (TODAES)

## REFEREED CONFERENCE PUBLICATIONS

**Acceldroid: Co-designed acceleration of Android bytecode** **February 2013**
International Symposium on Code Generation and Optimization (CGO)

**Improving compiler-runtime separation with XIR** **May 2010**
Conference on Virtual execution environments (VEE)

**Trace Based Compilation in Interpreter-less Execution Environments** **March 2010**
Technical Report No. 10-01, Donald Bren School of Information and Computer Science

**Phase detection using trace compilation** **Mar 2009**
PPPJ:International Conference on Principles and Practice of Programming in Java

**Array Reference Allocation Using SSA-Form and Live Range Growth** **Jun. 2000**
LCTES: Workshop on Languages, Compilers, and Tools for Embedded Systems

## INVITED TALKS

**Acceldroid: Co-designed acceleration of Android bytecode** **Dec. 2013**
University of California, Irvine *Irvine, California*

**Offloading Legacy Media filters to run on GPUs** **Nov. 2011**
Visual Processing Group - Intel Corporation *Santa Clara, California*

# ABSTRACT OF THE DISSERTATION

Just-in-Time Compilation Techniques for
Hardware/Software Co-Designed Processors

By

Marcelo Cintra

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2015

Professor Dr.Sc. Michael Franz, Chair

Recently, with the broad adoption of mobile devices, considerable research efforts have concentrated on innovative dynamic optimization techniques to improve the performance and energy efficiency of applications running on these resource-constrained devices, in order to meet the needs for better user experience, more functionalities, and reduced costs. In this dissertation, we explore the effectiveness, on performance and energy efficiency, of optimizations that are made possible when a Just-In-Time compiler can directly compile bytecodes to the internal, flexibly-designed, implementation ISA, rather than to the external, architectural ISA of a microprocessor.

We discuss the challenges and present the design and implementation of a novel acceleration framework to improve the performance-per-watt of applications written in modern object oriented languages that execute on managed runtime environments of mobile devices. Our acceleration framework allows the direct translation of bytecodes into the implementation ISA of modern co-designed processors, by a JIT compiler, bypassing the architectural ISA. We develop novel JIT speculative optimizations by leveraging both the semantically rich bytecode and the hardware features at the microarchitecture level, such as hardware support for: a) speculative execution, b) reduced indirect branches mispredictions, c) efficient call/return, and d) additional registers, which are not exposed by the microprocessor's architecture ISA for reasons of ISA compatibility.

We demonstrate the effectiveness of our acceleration interface in delivering increased performance-per-watt on mobile processors by providing an in-depth evaluation of two implementations: 1) first, using a real in-order co-designed processor; and 2) second, using a modern out-of-order co-designed research processor for the x86 architecture, running on a cycle accurate state-of-art simulator. Our framework provides the building blocks for the design of innovative hardware support and speculative optimizations to effectively deliver the stringent energy and performance requirements of applications running on mobile processors.

# Chapter 1

# Introduction

*"High thoughts must have high language."*

Aristophanes

Modern mobile devices combine the functionality of a pocket-sized communication device with PC-like capabilities. Almost all mobile devices have the ability to access a repository of thousands of applications, making their functionality almost limitless. Since the majority of these devices have a small form factor, the application performance and power consumption are critical aspects for delivering the best user experience. With the broad adoption of mobile devices, recent research has increasingly focused on techniques to provide rich user experiences and functionality for applications running on these resource constrained devices. This work focuses on improving the performance and energy efficiency of mobile devices to meet the user demands for more processing power, longer battery lifetime, new functionality, and reduced costs.

Improving battery life remains one of the biggest obstacles that mobile device makers have to deal with. Battery capacity is severely restricted due to constraints on the size and weight of these devices. Reducing power consumption has become one of the most critical design constraints for mobile devices. This makes the performance-per-watt - the number of computations that can be performed for every watt of power consumed (performance/power or power/performance) - a critical aspect since it directly correlates to the overall power budget and the computational capabilities of these portable systems.

As the capabilities of mobile devices increase, so does their need for more computing power. To help address these trends, recent research has focused on the design of energy efficient processors for mobile platforms [HS97, HS00, BO11] which can deliver high general purpose performance per watt of power consumed. This thesis focuses on improving the performance-per-watt of mobile applications written in managed languages, designed to enable platform-independent distribution of application software by execution on a Virtual Machine (VM). We propose a transparent scheme for hardware-accelerated bytecode execution to improve both the performance and energy efficiency of bytecode applications.

In the next two sections, we briefly highlight some characteristics of the bytecode applications that run on mobile devices and the challenges to running them on hardware (HW) accelerators. This overview helps us to understand both the key features of managed runtime environments and the feature-rich microarchitecture of codesigned processors. This understanding reveals the opportunity to re-purpose existing (and to design new) hardware features, at the microarchitecture level, for accelerating the runtime emulation engine and delivering improved power/performance efficiency for mobile applications, thus putting these ideas into context and motivating our work. We then introduce our approach for bytecode acceleration and summarize the contributions of this thesis.

## 1.1 Managed Programming Languages

For portability, mobile device applications are written in modern dynamic languages (e.g., Java, PHP, Python, JavaScript, C#, Perl, Ruby, etc.). These high level languages (HLL), also called managed programming languages (or just managed languages), are designed to be platform-independent by executing through a managed runtime environment. These high-level languages are distinguished from their low-level counterparts by abstracting away details of the underlying machine. That is why they provide portability, object-orientation, reflection, security, heap management, performance, and required garbage collection, at the cost of a small amount of overhead, freeing developers to concentrate on the business logic specific to their application.

These features, in addition to others also found in unmanaged languages, such as array bounds checking, null pointer checking and polymorphism, enable developers to create more maintainable applications, thus reducing development time. These features also have important security implications, since type-safe languages cannot be exploited through memory corruption. The support of these runtime services imposes a performance cost onto the managed language implementation. To compensate, high performance implementations of managed runtimes use just-in-time (JIT) compilation techniques to optimize for common case behavior while still retaining the programmability features. Another approach to mitigate the overhead is to use hardware acceleration.

Hardware acceleration is the use of specialized computer hardware to perform some functions faster than is possible using the general purpose architecture of a normal CPU. Hardware accelerators are designed for computationally intensive software, like graphics or video processing. It is challenging to run bytecode cost-efficiently on hardware (HW) accelerators, however. First, bytecode is general purpose in nature rather than the domain-specific code easily accelerated by a dedicated hardware accelerator (e.g., graphic code accelerated by a GPU). Second, for type safety and security, bytecode is mandated to run indirectly through a managed runtime (e.g., Java Virtual Machine) [LY99] rather than directly running on a hardware accelerator. Third, many features in bytecode are designed for Just-In-Time [DS84, HCU91, HU96, GPM+99] compilation in managed runtime, which is not efficient enough to directly run on a hardware accelerator.Furthermore, it is generally challenging to design an accelerator interface for a HW accelerator due to the complicated synchronization and communication between CPU and the accelerator.

Previous work [OT97, aS00] has attempted to implement a dedicated Java processor to execute bytecodes natively in silicon. The most referenced design is picoJava [DPC+00, OT97, Mic99], a Java processor specification released by Sun Microsystems. However, due to the aforementioned challenges to running bytecode in hardware, this processor was never released commercially. Prototype implementations [PS07, Got10], using FPGA, show some of the implementation challenges. However, due to their system limitations, they don't provide a sound evaluation with significant real world and benchmark applications. There have also been some other embedded Java processor implementations [aS00]; however, they are domain specific and have not become mainstream.

Since the bytecode was designed to run on a managed runtime, which relies on a dynamic compiler (JIT) to deliver high performance execution, we argue that a JIT compiler can benefit from the hardware features of a hardware/software codesigned processor to boost the performance (we also call bytecode acceleration) and energy efficiency of modern mobile processors, as we will show next.

## 1.2 Hardware/Software Co-Designed CPUs

For flexibility, modern CPUs decouple the architectural Instruction Set Architecture (ISA) from the internal ISA [Sla94] [Gwe95] in the micro-architecture design as shown in Figure 1.1 on the next page. We refer to this low-level internal ISA as the implementation ISA (I-ISA) to emphasize its implementation dependence, since it is not an existing ISA exposed to the user like the architectural ISA. For example, Intel CPUs implement an internal-ISA called micro-operation (Uop), which is completely transparent to x86 OS and applications. The architectural ISA code (i.e., x86 instructions) in the Operating System (OS) and applications is decoded into internal-ISA code by a hardware decoder before running on the CPU pipeline. Since the internal-ISA is transparent to the user, it can be flexibly designed to reach the power performance requirements of the design.

Hardware/Software (HW/SW) co-design techniques allow customized processor designs without concerning the backward compatibility of the architecture-ISA. The I-ISA of HW/SW co-designed processors can be flexibly and innovatively designed, based on the workloads and the power/performance budgets. Because the I-ISA is implementation dependent, the hardware and software along this boundary can be co-designed, blurring the division between the two. The resulting flexibility enables the distribution of power performance features between and across the hardware/software interface. This allows micro-architecture innovations on the CPU pipeline and corresponding internal-ISA designs in new generations of CPUs without affecting ISA backward compatibility. Legacy binary can directly run on new generations of CPUs even though the pipeline and internal-ISA designs are dramatically changed.

Examples of micro-architecture innovations in co-designed systems include: hardware support to reduce indirect branch mispredictions during the interpretation of cold bytecode [KN11], support for speculative execution [DGB+03, NRS+07, NDZ10], support for efficient execution of function calls and returns [KN11], support for efficient floating point operations [LGM+14, Cor13], and the provision of large internal register files to perform register promotion.

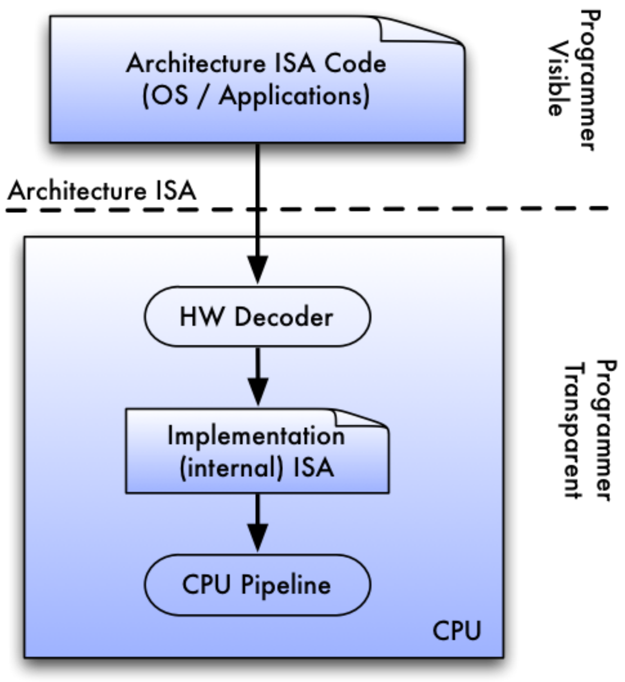


Figure 1.1: Modern CPU design

## 1.3 Problem Statement

The managed runtime of a HLL cannot take full advantage of the rich micro-architecture features of modern co-designed processors to optimize the power consumption effectively, for the following reasons: First, all the co-designed HW features at the micro-architecture level are tightly coupled with the internal pipeline implementation and cannot be exposed to the JIT compiler by architectural (x86) ISA for reasons of backward compatibility. Therefore, the JIT compiler cannot apply optimizations targeting these micro-architectural features because they are not exposed in the architectural ISA. Second, when the JIT compiler translates the bytecodes into the architectural ISA, it loses all the rich semantics of bytecode, which could, if preserved, be used by the Dynamic Binary Translator (or the hardware decoder for the architectural ISA) to perform more aggressive dynamic optimizations to leverage the co-designed HW features, and thus generate more power-efficient code. Lastly, the double translation of bytecodes: 1) first into architectural ISA by the JIT compiler, and 2) then from architectural ISA into the implementation ISA, by the Dynamic Binary Translator (DBT), results in higher overhead, reducing the performance and increasing the power consumption as the double translation requires more instructions to be executed in the processor pipeline. The framework for direct translation of bytecode (managed code in general) into the implementation ISA, proposed in this dissertation, aims at solving this problem.

## 1.4 Thesis statement

My thesis can be summed up in one sentence:

*This thesis explores the effects, on performance and energy efficiency, of optimizations that are made possible when a JIT can directly compile bytecodes to the internal, flexibly-designed, ISA rather than to the external, architectural ISA of a microprocessor.*

This dissertation presents a framework for direct translation of bytecode into the internal ISA of a microprocessor, which allows the development of JIT optimization techniques that benefit from the rich micro-architecture features not exposed by the external ISA for compatibility reasons. The proposed optimization techniques represent a significant contribution to the state of the art dynamic compilation techniques, which has been an area of intense research, to deliver the high demands for performance and energy efficiency of applications running in managed runtime environments of resource constrained devices, such as mobile devices.

## 1.5 Our Approach

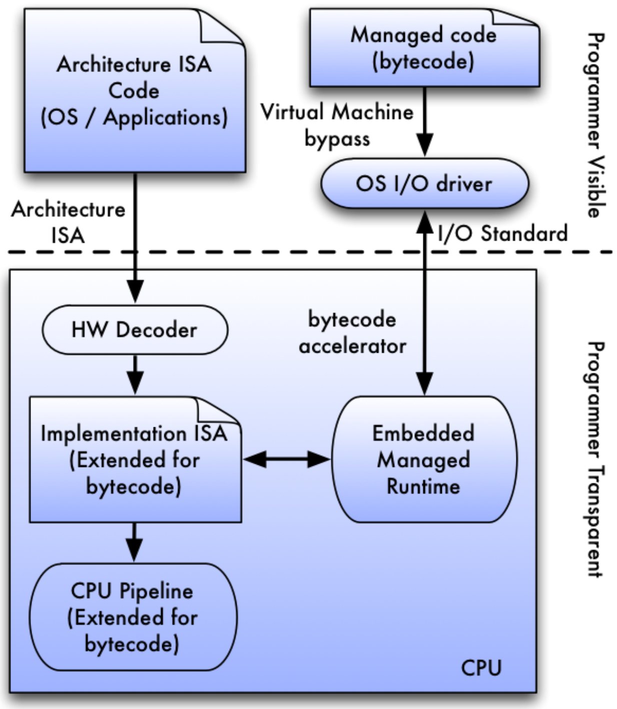


Figure 1.2: Bytecode Code Accelerator Design

Inspired by the transparent mechanism to translate architectural ISA into internal ISA [SCF+03, PL99], employed by modern CPU designs, we present a novel bytecode accelerator design as shown in Figure 1.2. From the programmer point of view, there is no difference between our bytecode accelerator and a regular hardware accelerator accessed through a HW I/O interface. An OS I/O driver is provided to manage the accelerator following certain HW I/O standards (e.g., PCI standard). The virtual machine (VM) passes the bytecode to the accelerator via the OS I/O driver. However, internally we do not actually implement a dedicated hardware accelerator to run the managed code. Instead, we develop an internal embedded bytecode runtime written in internal-ISA to intercept the I/O instructions from the OS I/O driver and emulate the accelerator execution of bytecode on the CPU pipeline. In this way: 1) we allow the development of new JIT optimizations to improve the bytecode efficiency through the embedded managed runtime, closely-coupled with the CPU pipeline, leveraging the internal micro-architecture extensions; 2) we allow micro-architecture extension of the internal-ISA and CPU pipeline design for accelerating the general-purpose bytecode execution transparently without any architecture-ISA change; 3) we avoid many issues in dedicated hardware accelerators such as long access latency, large die area, high cost, etc.; and 4) we allow the direct translation of bytecodes into the I-ISA, avoiding the double translation to architectural ISA and then to the I-ISA.

By providing a mechanism for direct translation of bytecode to the internal ISA, our technique allows the development of JIT compilation techniques to use the high level semantics of bytecode to perform aggressive optimizations, leveraging the rich micro-architecture features of a HW/SW co-designed processor as a means of hardware acceleration. Since the hardware used for bytecode acceleration in our scheme is implemented in the CPU pipeline (I-ISA), not in a separate unit from the CPU, we avoid the communication and synchronization complexities and overheads of hardware accelerators external to the CPU.

This dissertation will demonstrate that our methodology can deliver high application performance at a low power budget, and succeeds in dealing with challenges such as energy efficiency, cost, reduced die area, and complexity, meeting the emerging mobile devices needs for features and flexibility.

## 1.6 Contributions

This thesis makes the following contributions:

- The first contribution of this dissertation is a new interface to allow the direct translation of bytecode into the internal implementation ISA of a CPU, leveraging the hardware support at the micro-architecture level for the acceleration of bytecode efficiently and transparently. We identify interesting system issues and provide novel techniques for handling them. We also present a an extension of our interface to control the execution, profiling, and the performance evaluation of bytecode execution on a hardware simulator.

- We contribute novel JIT optimizations that leverage the HW/SW co-designed features to efficiently translate Dalvik bytecode. These optimizations are not easily applied by either the DVM or the existing DBT on co-designed systems.

- We demonstrate that our technique succeeds in delivering high performance-per-watt on co-designed processors, by providing an in-depth evaluation of two implementations of our bytecode acceleration scheme: 1) the first using a real system using a HW/SW co-designed processor, the Transmeta Efficeon; and 2) the second using a modern Out-Of-Order co-designed research processor for the x86 architecture, running on a cycle accurate state of art simulator.

- We show that our hardware-based bytecode accelerator, which is embedded in the CPU pipeline, does not require new architecture ISA instructions, separate co-processors, or hardware extensions to allow efficient alternation of bytecode execution between accelerated and non-accelerated modes. We leveraged existing hardware features exposed by the internal ISA in the two systems we evaluated, and we show how our scheme would benefit from additional hardware support designed specifically for bytecode acceleration.

- Finally, we show that our bytecode accelerator greatly improves the programmability compared to dedicated hardware accelerators with only minor changes to the user level virtual machine implementation for accelerating the bytecode execution.

We argue that the bytecode accelerator mechanism proposed in this dissertation, coupled with novel JIT optimization techniques, based on a HW/SW co-design methodology, is an effective approach for addressing the requirements of modern mobile CPU designs to deliver high performance and long battery life, while reducing die area, cost and complexity. Our techniques provide an entirely new approach to mobile processor design, by allowing innovations in the underlying microarchitecture to support the high level abstractions and services of modern managed runtimes executing on mobile devices, and thus demonstrating the benefits of microprocessors implemented as hardware-software hybrids.

To the best of our knowledge, no previous work has proposed a power efficient mechanism for bytecode acceleration that does not require complex hardware or new ISA instructions, and that has a simple and efficient interface to alternate execution between accelerated and non-accelerated bytecode execution mode, as in our approach.

## 1.7 Thesis Organization

This dissertation begins in Chapter 2 with an overview of virtual machines services provided by modern runtime environments, focusing on the opportunities for potential hardware assisted dynamic optimizations. We also present an informal survey on related codesigned mechanisms that: 1) allow co-designed processors to emulate full ISAs, and 2) provide real advantages over a hardware-only ISA implementation. We limit our explanation to the codesigned features that enable improved performance and power efficiency, specifically the ones we leveraged to accelerate bytecode execution.

In Chapter 3, we present AccelDroid, the bytecode acceleration framework we implemented in a state-of-art binary translation system, the Transmeta Efficeon. This chapter presents the accelerator design, the novel JIT optimizations we developed to leverage the co-designed features of Efficeon, and a comprehensive evaluation using the industry standard benchmark CaffeineMark 3.0.

In Chapter 4, we present a novel interface to control the execution and the performance evaluation of bytecodes on a hardware simulator. This scheme is based on the accelerator interface presented in Chapter 3, and shows that the implications of our research extend beyond bytecode acceleration, with direct application on simulation, performance monitoring, profiling and debugging. The description of this simulation control mechanism is provided as a secondary contribution of this dissertation, and it is used to evaluate the implementation of our bytecode acceleration on a modern codesigned processor simulator, which is the subject of Chapter 5.

Chapter 5 presents the implementation and evaluation of our bytecode acceleration on top of a state-of-art Out-of-Order(OOO) x86 processor simulator. The chapter shows the design of our acceleration scheme, and how it can deliver improved performance on a modern OOO execution engine with minimal (to no) increases in energy consumption.

Chapter 6 presents a rich outlook on future research.

The dissertation ends with conclusions in Chapter 7.

In the Appendices we show A) an overview of the Dalvik Virtual Machine (DVM), B) an overview of the Transmeta Codesigned Efficeon system, both used to implement the acceleration framework proposed in this dissertation, and C) and example of a direct translation of a bytecode trace into the internal ISA of the Efficeon processor.

# Chapter 2

# Related Work

*"We are products of our past, but we don't have to be prisoners of it. "*

Rick Warren

This chapter serves as a foundation for the topics discussed in subsequent chapters. This thesis combines advances in dynamic compilation systems and microarchitecture support to improve energy and performance efficiency of applications running on modern mobile devices. The relevant existing work related to this dissertation falls into the following two categories: 1) The dynamic compilation techniques used in the runtime of modern managed languages, and 2) the rich microarchitecture features of modern CPUs. In the earlier part of the chapter, we show an overview of the related research on dynamic compilation techniques aimed at improving the efficiency of the emulation engine of virtual machines. In the later part of the chapter, we show an overview of the design benefits of Hardware/-Software Co-designed processors, as well as the closely related work on Dynamic Binary Translation that either 1) made it possible the implementation of the acceleration techniques, or 2) gave us insights to better design the acceleration framework proposed in this dissertation.

## 2.1 Virtual Machines

Every computer platform provides an instruction set architecture (ISA) as a consistent interface to expose hardware functionalities to programmers, so that software can be used across a wide range of different processors implementing that ISA. Criteria such as efficient instruction decoding, code density, and backward compatibility with previous ISAs have dictated the design choices [Pat89, PD80]. Only recently have we witnessed the emergence of virtual ISAs (V-ISA) that have not been designed with direct hardware implementation in mind, which are semantically much richer than previous hardware-directed ISAs and require a managed runtime environment (a virtual machine) for execution.

For portability, modern languages such as Java and C# compile the application source into a high-level, byte-encoded intermediate representation (IR), not to a machine-specific code that could run directly on a computer. The IR is commonly called managed code, bytecode, or simply IR. In this dissertation we call the IR bytecode. The bytecode is designed to execute through a managed runtime environment, which enables the platform-independent distribution of application software. The bytecode is kept in a file along with metadata that describes its classes, methods, and attributes (such as security requirements). The semantically rich bytecode is one example of a V-ISA, which is commonly the unity of deployment of applications in mobile devices.

While frequently referred to as a language virtual machine, with the Java Virtual Machine (JVM) [LY99] and the Common Language Runtime (CLR) [ECM10] being the most familiar examples, the actual virtual machine component is but one element of a modern runtime environment. In addition to the virtual machine, which emulates the bytecodes of a program, today's runtime also provides a generalized way to handle the sophisticated features of object-orientation, security, heap management, garbage collection, networking, and performance, at the cost of a small amount of overhead, freeing developers to concentrate on the business logic specific to their application.

This dissertation aims at improving the energy and performance efficiency of the emulation engine, which we introduce in Section 2.2.3 on the next page, by better utilizing the hardware features only available at the micro-architecture level. Our framework can also be extended to support other runtime services, but that is the subject of future research.

## 2.2 Virtual Machine Services

### 2.2.1 Mobile Code Security

The bytecode verifier is one of the key components of a virtual machine security system. The verifier checks that the bytecode of downloaded applications is correct before execution. The verifier not only checks that bytecode is well formed, but it also ensures that no execution can violate any of the language typing rules. Bytecode verification has been studied extensively from a correctness perspective, and a large body of prior work exists that discusses how to ensure that bytecode verification is actually safe [Ler01, Gal06]. The bytecode verifier performs a data flow analysis [Qia00], tracking the values and their types along the edges of the Control Flow Graph (CFG), checking to ensure that no rules of the type system are violated. Gal [Gal06] demonstrated the need, when dealing with mobile code, for algorithms that are not only correct, but also efficient [GPF08]. Although the acceleration scheme proposed in this dissertation focuses on improving the energy and performance efficiency of bytecode execution, it can also be extended to improve the efficiency of the bytecode verifier.

### 2.2.2 Garbage Collection

Automatic dynamic memory management, also known as garbage collection, is indispensable in controlling software complexity since it increases programmer productivity and software quality. Objects that are no longer accessible, i.e., "garbage", can be collected and then reused for new objects. The garbage collector provides automatic management of the address space by seeking out inaccessible regions of that space (i.e., with no references pointing to them), and returning them to the free memory pool. When the VM is running low on memory resources or at periodic intervals, the garbage collector is invoked to find the inaccessible garbage objects in the global memory heap, collecting their memory resources for later re-use. We believe there are untapped opportunities to use hardware support to improve the garbage collection performance. Our mechanism can easily be extended to be used by the garbage collector, and it is an interesting research topic to be pursued in future research.

### 2.2.3 Bytecode Emulation

For more than 35 years, computer system researchers have been investigating dynamic optimization techniques as an effective means of boosting application performance in virtual machines. The techniques have spanned all aspects, including modifications on source code, interpreter optimizations, just-in-time compilation, and replacing software with hardware.

#### Interpretation

An interpreter is a computer program that directly executes, i.e., performs, instructions written in a programming, scripting language or in a platform independent representation, such as the Java bytecode, without previously compiling them into a machine language program. The interpreter is usually implemented following one of these strategies to execute the program:

- Parse the source code and perform its behavior directly.
- Translate the source code into some efficient intermediate representation which is then executed (by interpretation).
- Explicitly jump to a stored pre-compiled code made by a compiler (also part of the interpreter system), which will interpret the instruction/statement from the source program (or bytecode).

The third approach has been widely used, and it is the one adopted by the Android Dalvik Virtual Machine, which we used to implement the acceleration framework proposed in this dissertation. The performance overhead of interpreters is largely due to the cost of opcode dispatch, or fetching of the next instruction to execute, since it is commonly implemented by an indirect jump instruction. In this dissertation we present a new technique to reduce the interpreter's dispatch overhead, which we will show in  section 3.3.1 on page 39,

**Just-in-Time Compilation**

The performance of managed applications depends on the quality of the dynamic optimizations and code generation performed by the dynamic compiler. Dynamic compilers, or Just-in-time compilers (JIT), are thus a key component of a managed runtime. The JIT compiler is responsible for compiling bytecodes into native code, and for providing information about the execution stack that can be used to do root-set enumeration, exception propagation, and security checks. Since the compilation happens at runtime, the JIT can exploit the profile information from the current execution of a program, such as dynamic types, to find opportunities for better optimization, and thus is a significant advantage over traditional static compilers.

To achieve high application performance, the runtime dynamically profiles bytecode applications and adapts them through aggressive dynamic optimizations and compilation to the target CPU ISA. In the usual course of events, the runtime first loads the bytecode from class files and verifies it to make sure the instructions are safe to execute. Then it starts the emulation process either by directly compiling the bytecode to the target ISA, or by interpreting it. In either case, the runtime profiles the execution in order to detect hot regions of execution for dynamic compilation to the target CPU ISA. Whenever a hot region of code execution is detected, the JIT compiler is called to perform aggressive optimizations, which produces a highly optimized machine code for the hot region. The runtime caches this optimized machine code to be used the next time the emulation engine reaches the start of the hot region just compiled.

The VM's JIT compiler takes advantage of the fact that the program execution time is spent in a very small part of the program, also known as the 90/10 law [HP03]. As such, the JIT compiler focuses its optimization effort by emitting optimized machine code specialized for the frequently executed regions of a program. The frequently executed regions form the unit of compilation of a JIT compiler, which can be represented as a method, a region [SYN03], or a trace.

Dynamo [BDB00] was the first trace-based optimizing compiler. A trace is a fragment of hot execution paths that can span beyond a basic block [GPF06a, GF06, BCW+10, HM11]. The traces are formed out of dynamic instruction streams by a binary interpreter. Dynamo pioneered many early concepts of trace selection and trace runtime management. A trace-based JIT works best over a wide range of memory and power-constrained portable devices, as shown in [BB09]. Our acceleration framework was implemented using a trace-based JIT compiler as we will see in Section 3.2.3 on page 37.

Recently Wimmer et al. [WCB+09] used traces to perform phase detection in a Java. A program phase is identified when the created traces are stable (i.e., there is a low trace exit ratio). Whenever the program execution start to take side exits from a trace, the program is said to be unstable ( i.e. between phases). In our acceleration framework, whenever a program enters an unstable phase, the runtime discards the unstable translated traces, and triggers the formation of a new trace once the program returns to a stable phase of execution.

In this dissertation, we present new techniques for enhancing the performance of application programs and the VM services related to the emulation engine, which are: dynamic profiling, interpretation, and Just-In-Time (JIT) compilation of bytecode.

## 2.3 HW/SW Codesign Methodology

Hardware/Software co-design is the partitioning and design of an application in terms of fixed (hardware) and flexible (software) components. The flexible components include programs written in high-level languages, assembly languages or in bitstreams. The fixed part consists of programmable components written using digital circuits created using word level combinatorial logic and flip flops. These circuits are usually modeled with building blocks such as registers, adders and multiplexers. A computer architecture designer must decide which functions are to be performed strictly under hardware control and which functions are to be performed by software [Man72, HJB+82].

The classic argument in favor of dedicated hardware design has been increased performance, resulting in more work done per clock cycle. This is due to the fact that a specialized hardware implementation can execute many operations in parallel. However, the increased performance is obtained by reducing the flexibility of an application. On the other hand, software excels over hardware in the support of application flexibility, and in reducing design complexity. A key observation is that there is a tradeoff between flexibility and efficiency.

Henessy et al. [HJB+82] argues that that architecture/compiler/system tradeoffs are inevitable, and that by correctly making design choices in all three areas the designer can obtain increased performance and reliability for lower cost. Lee et al. [LvDDM06] demonstrate the benefit of additional hardware support on bytecode applications performance on two computer architectures, the x86 IA32 and PowerIC. Their empirical results show better speedup on PowerPCs than on IA32s, due to the smaller number of registers on the IA32 compared to the PowerPC.

Similarly, we argue that there are compelling reasons for using a co-designed approach for mobile processor design, since the design of specialized hardware gives more opportunities to achieve better performance, lower power consumption, lower design complexity, better design customization, backward/forward ISA compatibility, or a combination of these. The techniques proposed in this dissertation provide further possibilities for leveraging specialized hardware for improved energy efficiency and performance. A co-designed approach has several advantages over a traditional hardware-only implementation, as we show in the next section.

### 2.3.1 HW/SW Co-design Advantages

Today's ISAs are valuable interfaces, ones in which huge software and infrastructure investments have been made. Compatibility has become the largest obstacle to implementing new ISAs that are better suited to today's technology, especially in mobile devices. Fortunately, co-designed processors, also called co-designed virtual machines [SSHB98], permit the development of new ISAs by enabling a different approach to general-purpose processor design, while still maintaining the compatibility of the architecture ISA that they implement.

Traditionally, the control and functionality of the ISAs have been implemented with hardware logic gates and memory. A hardware implementation of an ISA is often presumed to provide higher performance than a software implementation, but it lacks the flexibility that a software implementation has. HW/SW co-designed processors have emerged as an alternative for a hardware-only ISA implementation that can: 1) be used in a whole line of mobile devices aiming at different price/performance points, and 2) enable longer battery life by dynamically adjusting operating frequency and voltage to match the performance requirements of each device/application.

The co-design software layer comprises a virtualization machine monitor and a binary translation and optimization system. The co-design hardware supports a new private ISA. Both together are designed to meet the target power, performance and efficiency goals of the system. The software layer translates implementation ISA (or other ISAs) to the target private (internal) ISA. This design approach offers optimization headroom and flexibility not possible in hardware alone, while yielding a smaller core with a reduced power profile. The software layer requires memory for running as well as a place to put translated code for execution. Recent processor designs incorporate memory controllers on-die and make it relatively easy to reserve a sufficient amount of private memory to cache translated code.

Below, we list some important advantages of co-designed processors, and how they correlate to the bytecode acceleration techniques we propose in this dissertation:

- **Performance.** It allows for dynamic adaptation of the applications code to the underlying hardware. Dynamic compilation can exploit run-time profiling information and make traditional optimizations much more effective. This feature is probably the central piece of all of the techniques we propose in this dissertation. Our acceleration framework allows the dynamic adaptation of bytecodes to the underlying hardware.

- **Power and area.** One of the goals of the co-design process is the flexibility to move functionality to software, which can result in fewer logic transistors required to achieve the same performance, resulting in lower power and lower die area. This allows for a design that saves both die area and power. Our accelerator taps opportunities to better explore the available transistors in a co-designed processor so that they can be used to deliver higher power/performance efficiency to bytecode applications, by moving the implementation of rarely used architectural ISA instructions to software. We believe that this feature is of great value for embedded and wearable systems.

- **Custom Designs.** The co-design approach gives more flexibility to design processors for different markets, by making different trade-offs between what is implemented by hardware and what is implemented by software. This flexibility allows quick custom processor designs to the diverse ecosystem of mobile devices, making this approach more attractive than in the past. The custom design is one of the key features that enabled the implementation of our acceleration mechanism.

- **Backward compatibility.** New processor designs commonly have the requirement to run the vast pre-existing legacy code. For a processor manufacturer, this allows easier entrance into new markets and richer experiences for users than the alternatives. However, legacy code also imposes serious burdens for new designs. A co-designed core can achieve full ISA compatibility with far less legacy burden on the hardware. This is an important feature for mobile devices, since they have very short release cycles. Our scheme does not affect the backward compatibility of co-designed processors, as we will see in Chapter 3 on page 29.

- **Forward compatibility.** A co-designed processor can use new hardware capabilities without the need to recompile the original binary. The runtime software will adapt/optimize the binary to make use of these new features. In many situations, the source code for legacy applications is not available to be recompiled to take advantage of new ISA extensions. Since we compile bytecode, our bytecode accelerator runtime can benefit from new ISA extensions by dynamically compiling bytecodes to the internal ISA.

- **Prototyping.** The co-designed approach allows for the experimentation of new hardware features before making them visible to the end-user. These features can also be just made visible initially for internal experimentation, in order to evaluate and tune them before incorporating into products distributed in the market. This capability is highly desired and frequently used by mobile device manufacturers. For example, when Google first released the Dalvik JIT compiler, it was disabled by default.

### 2.3.2 ISA Emulation

Instruction set emulation is the process of implementing one instruction set, the *target* instruction set, to reproduce the behavior of software compiled to another instruction set, the *source* instruction set [SN05]. For example, Apple Rosetta [App06, Tra06] is an emulation platform for Mac OS that allows PowerPC applications to run on Intel-based Macintosh computers without modification. Apple released Rosetta when it transitioned the Macintosh platform from the PowerPC to Intel processor instruction set architecture, back in 2006.

Instruction set emulation is the key aspect of a HW/SW co-designed processor implementation, since it supports a program binary compiled for an instruction set that is different from the one implemented by the host processor. In this dissertation, we used the Transmeta Efficeon processor, which emulates 32-Bit x86 ISA on a very-long-instruction-work (VLIW) ISA processor. VLIW processors are a particular type of superscalar processors,

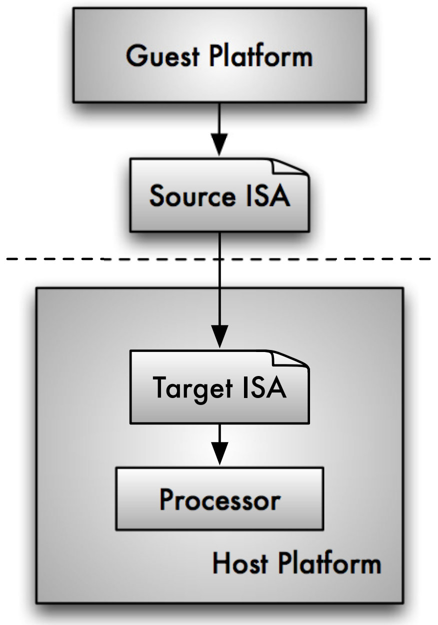


Figure 2.1: Emulation Process

since they can process multiple instructions in all pipeline stages and therefore can achieve a throughput higher than 1 instruction per cycle for some codes.

The instruction set emulation engine can be carried out using two emulation methods: interpretation or dynamic binary translation [Die98, Pro01, EA97], each one requiring different amounts of computing resources and offering different performance and portability characteristics. Since efficiency is a crucial aspect of an instruction set emulator, for optimal performance the emulation engine typically performs dynamic binary translation, on-the-fly, while the program is executing.

### 2.3.3 Dynamic Binary Translation

Binary translation is the process of translating machine code binaries from one instruction set architecture to another. It can be done statically, by translating a whole binary to a new binary for the target platform, or dynamically, by translating code on the fly. Most binary translation systems currently in use are migration tools which aid the transition from old architectures to newer ones.

The Rosetta emulation platform uses a dynamic binary translator, which is directly integrated into the Operating System, and transparently converts PowerPC instructions into IA-32 instructions and optimizes them [App06, Tra06].

## 2.4 HW/SW Co-Designed Processors

HW/SW co-designed processors such as the DAISY processor [EA97], the Binary-translation Optimized Architecture (BOA) [GAS+00], the UQDBT [UC00], and the Transmeta Efficeon [Kre03, DGB+03] use Dynamic Binary Translation [SCK+93] technology to translate the code of the architectural ISA into the code of the internal customized ISA implementation in order to achieve increased power/performance efficiency. However, such processors cannot directly translate bytecode in dynamic languages into the internal ISA or leverage unique co-designed HW features, which are not supported in the architectural ISA. Architectural ISA extension for bytecode raises concerns related to ISA backward compatibility because the bytecode standard may change independent of the ISA. Our acceleration scheme provides the benefits of HW/SW co-design for bytecode without any architectural ISA modifications. Modern x86 implementations decode instructions into one or more micro-operations in order to deal with the complexity of the ISA [SCF+03, PL99], but still preserves the architectural ISA for backward compatibility.

Existing JIT compilation techniques [Fra94, DS84, SYK+01, SYK+05] translate portable bytecode into the code of the architectural ISA to allow it to run on various platforms. However, when the unique features of HW/SW co-design are not properly exploited, the runtime translation/emulation overhead reduces the performance of bytecode execution to below that of the execution of the native code. Most dynamic binary translation/optimization systems, including Dynamo [BDB00], IA32-EL [BDE+03], DynamoRIO [BGA03], Pin [LCM+05] and HDTrans [SSNB06], operate on the level of the architectural ISA and do not utilize the specialized features of HW/SW co-design, and thus suffer a slowdown similar to that of the JIT compilation. Our scheme allows for the translation of bytecode directly into the code of the internal ISA implementation and leverages both the high-level bytecode information and the unique features of HW/SW co-design in order to reduce the translation/emulation overheads and improve the performance of the emulated execution. Our scheme can also be extended to leverage annotation-based information embedded in the bytecode [ANH99, KC01], which can be produced by static analysis or by profile-based information available at the virtual machine runtime. Our framework also leverages the rich dynamic profiling information provided by the Efficeon processor, and performs continuous program re-optimizations [KF01] for power performance improvements.

Jazelle DBX (Direct Bytecode eXecution) [ARM14] uses hardware to directly execute Java bytecode. The hardware execution of bytecode requires additional die area and is inflexible with respect to modifications. For example, Jazelle DBX is designed for Java bytecode, and unless the hardware is redesigned, it cannot execute Dalvik bytecode. AccelDroid can be easily adapted to accelerate both Java bytecode and Dalvik bytecode without any hardware changes on the Efficeon machine. Jazelle RCT (Runtime Compiler Target) adds new ISA extensions for JIT compilation, in which the change in the ISA is visible to the OS/applications. Our HW/SW co-designed accelerator transparently explores the internal co-designed HW support without exposing any new ISA support to the OS/applications. This allows for efficient hardware and internal ISA design without affecting the ISA backward compatibility, which is important for x86 processors. Furthermore, the co-designed HW features that we leverage to accelerate Dalvik bytecode, e.g., LINKPIPE, FLOOK, and SPECULATION, are very different from the new instructions used in Jazelle RCT.

### 2.4.1 Hardware Support in Co-Designed Processors

This dissertation relates to several previous research studies on specialized hardware support for the purpose of enabling efficient execution of the translated code on hardware/-software co-designed systems.

Previous work explored the use of atomic regions [PTBC00, DGB+03, NRS+07, NDZ10] as a means of increasing the dynamic length of branch-less regions of instructions for the purposes of dynamic speculative optimizations. The region is considered atomic because if one instruction in the region is committed to an architectural state, then all instructions are committed. Branch-less regions are formed by replacing original branch instructions with assertions. When an assertion fires (or an exception such as a TLB miss occurs) during the atomic region execution, the hardware must roll the architectural state back to the beginning of the atomic region, and redirects the control flow to the original address for the instruction at the beginning of the atomic region [FBC+01]. This recovery mechanism enables the optimizer to make speculative optimizations without the necessity of generating recovery code, potentially increasing the aggressiveness of the optimizations. Similarly, our acceleration scheme leverages the hardware recovery mechanism to recover the architectural state in the event that assumptions made during bytecode optimization - such as an assumption that a pointer is non-null or that an array is in-bounds - become invalid during execution.

In TAO [BWW+10], larger regions of code for compilation can be formed by using a two-level atomicity support. The system performs speculative checkpoints for each loop iteration and executes until it runs out of resources. Whenever the system runs out of resources, the execution performs a short rollback to the last speculative checkpoint (e.g., discarding the data produced by the last loop iteration) and commits the state. The related work presented in LAR-CC [BWBW11] prevents the overhead caused by these short rollbacks by predicting if the system will run out of resources while dispensing the two-level hardware atomicity support.

The hardware recovery mechanisms of the Transmeta Efficeon [Kre03] included a 64KB L1 speculative data cache and a 32 entry victim cache, capable of holding a larger amount of speculative data. Efficeon's DBT translator conservatively inserts commit operations in the middle of a translated region (e.g., loop headers) to prevent the execution from running out of resources, reducing the size of the atomic regions. Efficeon also has the link pipe support necessary to reduce indirect branches mispredictions [KN11], and a fast lookup (flook) [KN11] feature for the efficient translation of function calls and returns.

Other examples of hardware support to enable efficient execution of translated code include: self-modifying code detection [DGB+03], indirect branch execution [KS03], and memory alias detection [Kla00a], which are all examples of mechanisms that rely on hardware support to enable the efficient execution of translated code.

This dissertation explores the effects, on performance and energy efficiency, of optimizations that are made possible when a JIT can directly compile bytecodes to the internal, flexibly-designed, ISA of co-designed processors. We develop new JIT optimizations to take advantage of the aforementioned microarchitectural hardware support in co-designed processors to improve the performance-per-watt of mobile applications, as we will show in the next chapters.

# Chapter 3

# Bytecode Acceleration Techniques

*"An ounce of performance is worth pounds of promises."*

Mae West

In this chapter we present AccelDroid: a framework for bytecode acceleration of Dalvik bytecodes on a real system. We used the Android software stack running on top of a HW/SW codesigned machine, the Transmeta Efficeon. We highlight key design choices as well as the changes to both the Dalvik virtual machine and the Code Morphing Software in order to implement AccelDroid. At the end of the chapter we present the evaluation of our scheme.

## 3.1 Bytecode Acceleration with AccelDroid

In this section, we present AccelDroid [WCW13], which is an acceleration interface for the execution of Android Dalvik bytecode on an HW/SW co-designed processor through direct bytecode translation in the DBT without the need for any change in the architectural ISA.

Figure 3.4(b) illustrates the execution of bytecode through AccelDroid, in which a virtual accelerator interface is designed to pass the bytecode directly to the internal DBT, bypassing the Dalvik Virtual Machine (DVM) runtime. From a programmer's point of view, there is no difference between our managed code accelerator and a regular hardware accelerator accessed through an HW I/O interface. The OS I/O driver for managing the bytecode accelerator is implemented following certain HW I/O standards (e.g., the PCI standard). To accelerate bytecode execution, the DVM passes the bytecode to the accelerator via the OS I/O driver. To the DVM and the OS, AccelDroid operates in the same manner as a bytecode accelerator that is controlled through standard I/O instructions. However, internally, we do not implement a dedicated hardware accelerator to run the bytecode. Instead, the accelerator is implemented virtually by a bytecode-aware DBT, which virtualizes the I/O instructions and directly translates the bytecode into the code of the internal ISA implementation for efficient execution.

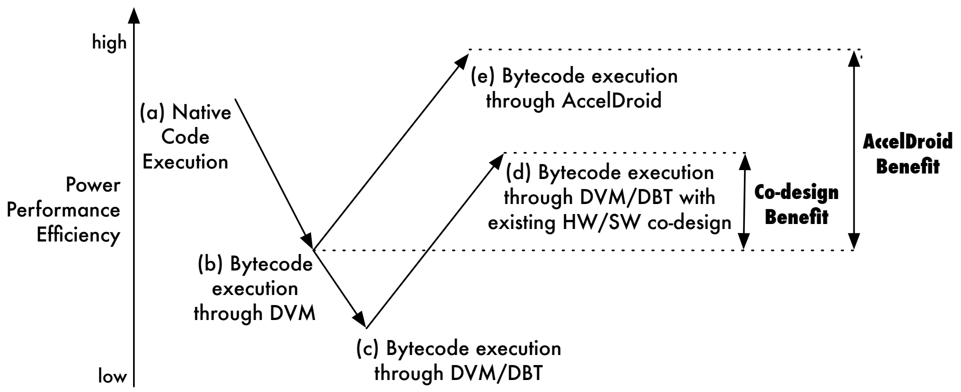

high
low
Power Performance Efficiency
(a) Native Code Execution
(b) Bytecode execution through DVM
(c) Bytecode execution through DVM/DBT
(d) Bytecode execution through DVM/DBT with existing HW/SW co-design
(e) Bytecode execution through AccelDroid
Co-design Benefit
AccelDroid Benefit

Figure 3.1: The Benefit of Using AccelDroid for Bytecode Execution

Figure 3.1 illustrates the advantages of AccelDroid. On normal processors (i.e., processors without HW/SW co-design), the energy/performance efficiency of bytecode execution through the DVM (see Figure 3.1 (b)) is typically worse than that of execution of the native code (see Figure 3.1 (a)) because of the translation and emulation overheads incurred in the DVM. On HW/SW co-designed processors, the efficiency of bytecode execution through the DVM and, subsequently the DBT, is even further degraded because of the further translation and emulation overheads incurred in the DBT (see Figure 3.1 (c)). However, HW/SW co-design techniques can improve the energy/performance efficiency with respect to normal processors (see the co-designed benefit noted between Figure 3.1 (d) and Figure 3.1 (b)).

The efficiency of bytecode execution through AccelDroid (see Figure 3.1 (e)) is further improved for two reasons. First, the bytecode is translated only once, by the DBT, instead of twice, once by the DVM and once by the DBT. Moreover, AccelDroid enables close co-design of the DBT SW and the internal HW, allows bytecode-specific translation and optimization to be performed while leveraging the higher-level semantics of the bytecode, and improves the efficiency of bytecode execution even with respect to the execution of the native code (see Figure 3.1 (a)). As a result, the benefit achieved through AccelDroid (i.e., the difference between Figures 3.1 (e) and (b)) can be significantly greater than the co-design benefit (i.e., the difference between Figures 3.1 (d) and (b)).

To demonstrate the advantages of AccelDroid, we implemented AccelDroid on the Transmeta Efficeon [Kre03], an HW/SW co-designed processor that translates x86 code into the internal Efficeon code. We developed several novel optimizations of AccelDroid to efficiently translate the bytecode directly into the internal Efficeon code, thereby leveraging the existing internal co-designed HW features in the Efficeon. First, we leverage the internal LINKPIPE feature [KN11] in the Efficeon to reduce indirect branch mispredictions during the interpretation of cold bytecode. Second, we leverage the internal SPECULATION feature [DGB+03, NDZ10] in the Efficeon to enable efficient bytecode null-pointer and array-bound checks. Third, we leverage the internal FLOOK [KN11] feature in the Efficeon for the efficient translation of bytecode function calls and returns. Finally, we leverage the large internal register file in the Efficeon for efficient bytecode register promotion. All these co-designed HW features are tightly coupled with the internal pipeline implementation in the Efficeon and cannot be exposed to the JIT compiler at the x86 ISA level for reasons of backward compatibility. Therefore, the DVM cannot apply our optimizations because of the lack of these co-designed HW features in the x86 ISA. The existing DBT on the Efficeon machine (also called CMS, or Code Morphing Software [DGB+03]) also cannot apply these optimizations because of the loss of bytecode-specific information that occurs once the DVM has translated the bytecode into x86 code.

## 3.2 AccelDroid Infrastructure

Figure 3.2 on the following page depicts the overall Android software stack. Android applications are developed in the Java™ language, whose source code is first compiled into Java bytecode and then converted into Dalvik bytecode through the Android SDK. These applications can invoke the bytecode in the Application Framework for system services such as input and display. All bytecode in *Applications* and the *Application Framework* must run through the *Dalvik Virtual Machine* (DVM) in the *Android Runtime*. The bytecode can also invoke the native x86 code in the *Libraries* through *Java Native Interface* (JNI) calls.

### 3.2.1 OS Acceleration Driver

We designed AccelDroid to directly access and update the CPU architecture states, including general-purpose registers, process virtual memory, etc. In order to flexibly support the Dalvik Virtual Machine, we have defined an Acceleration Control Block (ACB) data structure that specifies the mappings from all of the bytecode states to the CPU architecture states such that the accelerated bytecode execution can access and update the corresponding CPU architecture states. For example, if the ACB maps the bytecode register r0 to the x86 register EAX, then the bytecode instructions that update r0 will update EAX in the

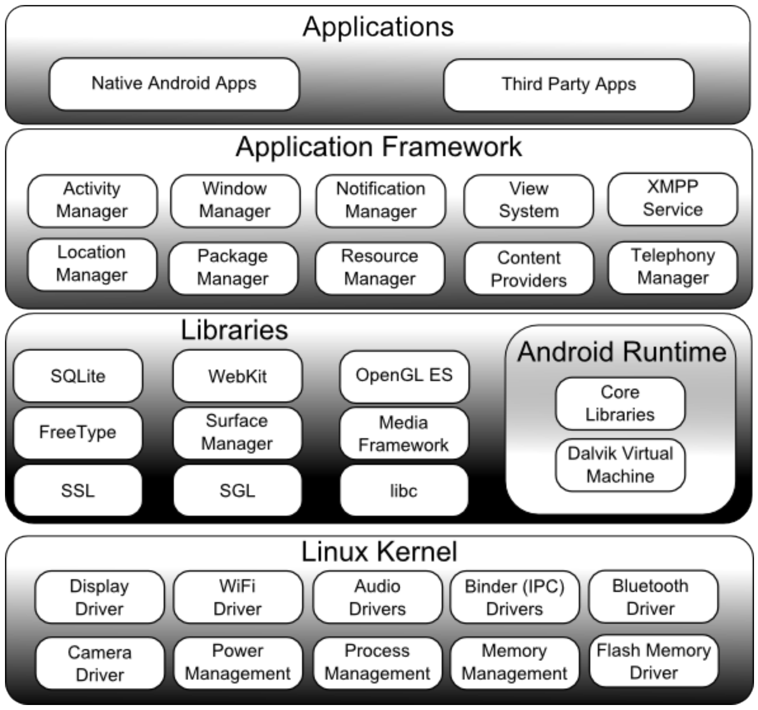


Figure 3.2: Android Software Stack

accelerated bytecode execution. The precise CPU architecture states controlled by ACB are maintained only at the moment of switching between architecture ISA code execution and accelerated bytecode execution. So the DBT can still freely run the bytecode with its internal microarchitecture states independent of ACB by just copying-in/copying-out the precise CPU architecture states according to ACB during the switching between architecture ISA code execution and accelerated bytecode execution.

We implement the OS acceleration driver as shown in the flow chart in Figure 3.3 on the next page. The acceleration driver interacts with the DBT runtime with privileged architecture I/O instructions, which trigger an internal microarchitecture trap to the DBT runtime execution (see details in Section 5.4.1 on page 93). During the OS boot time, the acceleration support is probed using architecture I/O instructions following standard interfaces (e.g., PCI interface) and the acceleration driver is loaded and initialized accordingly. The acceleration driver initialization reserves I/O address space and MMIO address space from OS/BIOS and sends them to the DBT runtime. After that, the acceleration driver can interact with the DBT runtime with architecture instructions accessing the reserved I/O or MMIO address according to our acceleration interface.

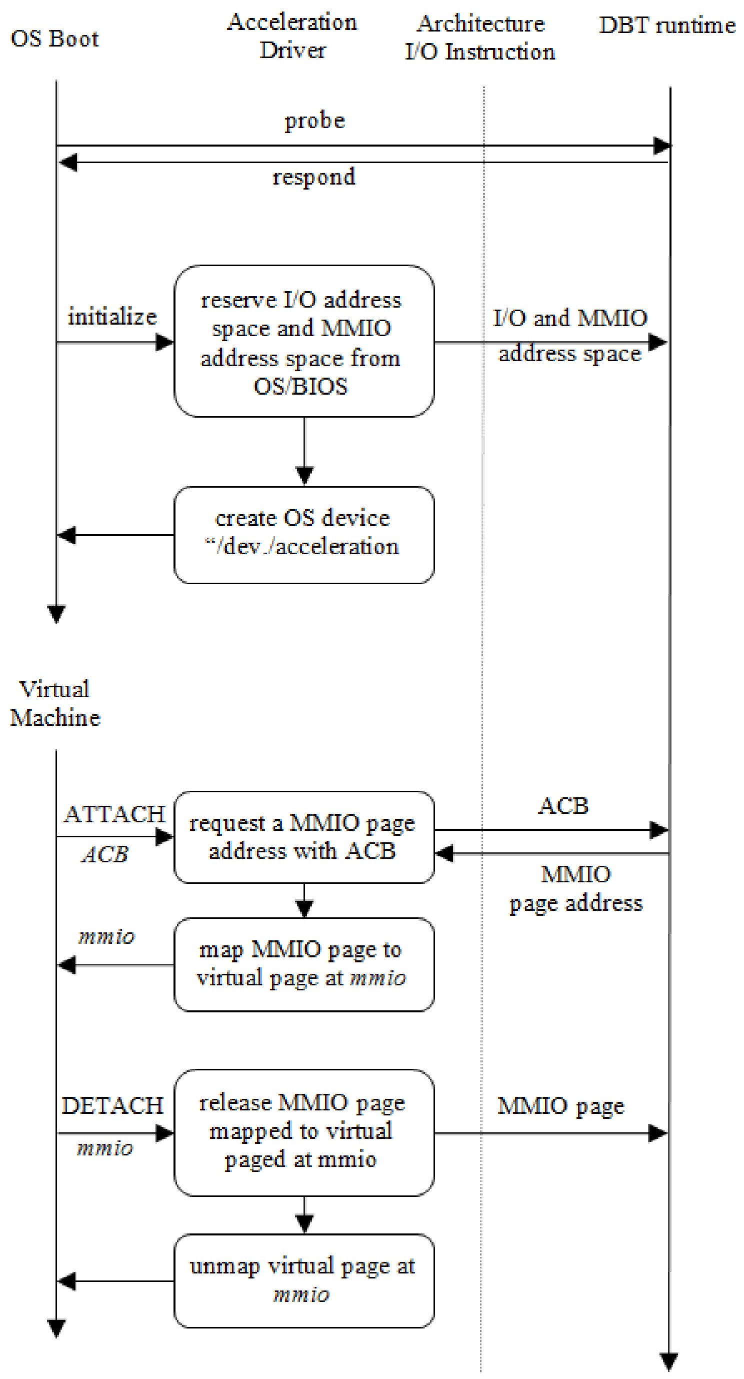

OS Boot
Acceleration Driver
Architecture I/O Instruction
DBT runtime
probe
respond
initialize
reserve I/O address space and MMIO address space from OS/BIOS
I/O and MMIO address space
create OS device "/dev./acceleration
Virtual Machine
ATTACH
ACB
request a MMIO page address with ACB
ACB
MMIO page address
mmio
map MMIO page to virtual page at mmio
DETACH
mmio
release MMIO page mapped to virtual paged at mmio
MMIO page
unmap virtual page at mmio

Figure 3.3: Flow Chart for the Acceleration Driver

When a virtual machine instance is attached to the acceleration interface with ioctl calls (see virtual machine implementation in  Section 5.3.1 on page 91), the acceleration driver will request a MMIO page address from OS and send an ACB to the DBT runtime through an architecture I/O instruction. The DBT runtime will respond with a MMIO page address from the available MMIO address space, using a different page for a different virtual machine instance attached to it. Since the user-level virtual machine has no privilege to directly access the physical MMIO page, the acceleration driver needs to map the physical MMIO page to a process virtual page and then return the virtual page address mmio to the virtual machine. This mechanism prevents different virtual machine instances from interfering with each other because a virtual machine can only access the MMIO page mapped to its process virtual space. The detachment from the acceleration interface will lead to release of the MMIO page for reuse.

### 3.2.2 Modifications to the Android DVM

We slightly modify the existing DVM implementation to accelerate the execution of bytecode through AccelDroid. Listing 3.1 shows the pseudocode for the existing DVM implementation. The bytecode is executed by DVM_run, through either interpretation or translation, until the end of the bytecode is reached. For AccelDroid, we implement an OS device driver to control the AccelDroid accelerator and modify the DVM implementation to run bytecode through the AccelDroid accelerator, as shown in Figure 3.2 on the following page. DVM first checks the availability of the AccelDroid accelerator by opening the corresponding OS device (i.e., /dev/dalvik) implemented by the device driver. If the AccelDroid accelerator is not available (i.e., the processor does not feature HW/SW co-design or AccelDroid support), it will run all bytecode in the same manner defined in the existing DVM implementation. Otherwise, the bytecode will be passed to the accelerator and executed through AccelDroid (see AccelDroid_run ).

To avoid expensive OS calls to the device driver when accessing the accelerator, Accel-

```
DVM() {
     Initialize the Dalvik Virtual Machine state DVM_state
     DVM_run(DVM_state)
}

DVM_run() {
     Execute bytecode in DVM_state through
                    interpretation/translation until the end of the bytecode
}
```

Listing 3.1: Existing DVM Implementation

```
DVM() {
     Initialize the Dalvik Virtual Machine state DVM_state
     // check AccelDroid accelerator
     if(AccelDroid_fd = open('/dev/dalvik', ...)) {
         AccelDroid_run(AccelDroid_fd, DVM_state);
     close(AccelDroid_fd);
    } else {     // no AccelDroid accelerator
         DVM_run(DVM_state);
    }
}

AccelDroid_run(AccelDroid_fd , DVM_state) {
     // obtain memory—mapped I/O address for AccelDroid
     AccelDroid_mem = ioctl(AccelDroid_fd, GET_MEM_IO, ...);
     while (not end of bytecode) {
           // bytecode execution through AccelDroid
           (*AccelDroid_mem)(DVM_state);
           // handle OS—dependent bytecode, HW faults/interrupts
           Interpretation of one bytecode in DVM_state
     }
}
```

Listing 3.2: AccelDroid DVM Implementation

Droid_run establishes a memory-mapped I/O address AccelDroid_mem through an ioctl (input/output control) call to the device driver. The bytecode in DVM_state can subsequently be executed simply through an indirect function call to the memory-mapped I/O address AccelDroid_mem. Internally, the DBT virtualizes the accelerator and intercepts the function call to the memory-mapped I/O address for bytecode translation/emulation.

### 3.2.3 Bytecode Acceleration in the DBT

The acceleration runtime is closely integrated with the DBT on the Efficeon, as shown in 3.4(b); this allows it to leverage the existing DBT facilities. Among them are the runtime profiler, translation cache management, the translation lookup and dispatch facility, chaining facilities, memory management, and exceptions and interrupts handling.

The existing DVM relies on OS support to execute several OS-dependent bytecode instructions. For example, the "new-instance" bytecode instruction may require OS calls to allocate new memory. Because the DBT in an HW/SW co-designed system runs below the OS and is independent of the OS (see Figure 3.4 (b)), it cannot rely on OS support for bytecode execution. Thus, AccelDroid can only execute OS-independent bytecode instructions. For OS-dependent bytecode instructions, AccelDroid execution stops and returns to the DVM. Then, the OS-dependent bytecode instructions will run through interpretation in the DVM (see "*Interpretation of one bytecode*" after the AccelDroid_mem call in Listing 3.2). Our experiments on a suite of CaffeineMark 3.0 benchmarks indicate that more than 99% of bytecode instructions are independent of the OS.

For OS-independent bytecode, AccelDroid follows the typical 2-phase translation and optimization technique [BDB00]. In the first phase, the bytecode is interpreted and profiled to identify the most frequently executed bytecode traces. As in most regular VM runtimes, our embedded bytecode runtime just-in-time compiles the hot bytecode traces into the internal VLIW ISA code (producing what we call translations) and stores them into the DBT translation cache so that future executions of the same bytecode do not require recompilation. In Section 3.3 on page 39, we present the new optimizations implemented in our acceleration JIT compiler.

The execution of bytecode through the DBT may encounter HW interrupts or trigger HW faults, which will require OS interrupt/fault handling. For OS transparency, we designed

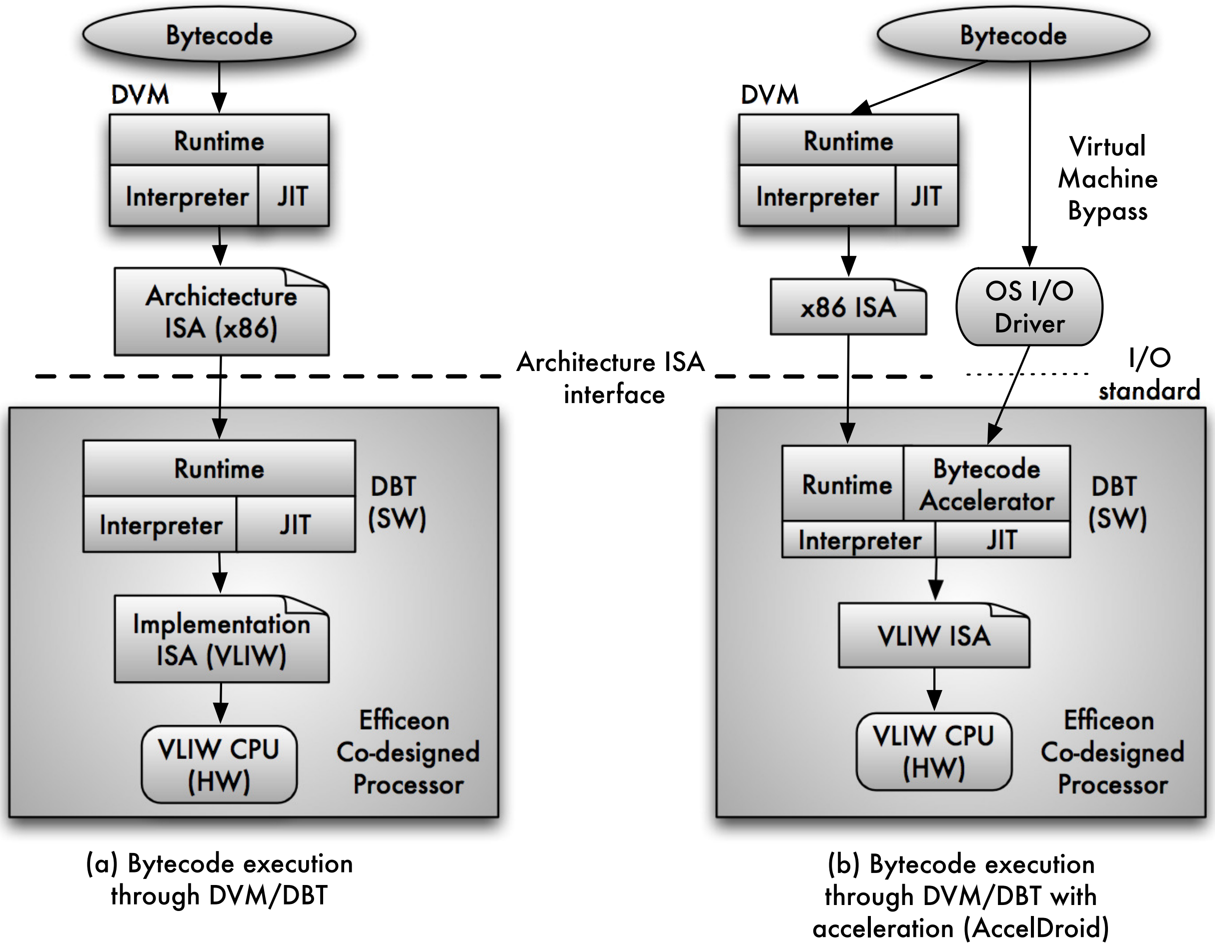


Figure 3.4: Bytecode Execution through a regular VM and through AccelDroid

AccelDroid such that it never raises exceptions. All HW interrupts/faults that are related to bytecode execution are caught along with the precise virtual machine state at the bytecode instruction boundaries by the DBT runtime. When the DBT runtime catches an interrupt or fault, it stops the bytecode execution in the precise state that existed before the bytecode instruction that raised it. The execution is then switched to the DVM in the architectural (x86) ISA, which will then re-execute the bytecode that raised the interrupt or fault. Thus, HW interrupts/faults will be handled by OS interrupt/fault handlers during the interpretation of the bytecode in the DVM.

```
Interpreter:
  fetch a bytecode b
  jump to handlerTable[b.opcode]

OP1_handler:
  ...   // interpret OP1
  fetch next bytecode nb
  jump to handlerTable[nb.opcode]

OP2_handler:
  ...   // interpret OP2
  fetch next bytecode nb
  jump to handlerTable[nb.opcode]
      ...
```

Listing 3.3: DVM Interpreter

## 3.3 Bytecode Optimizations

AccelDroid implements a set of novel bytecode optimizations. All of the proposed optimizations leverage both bytecode-specific information and the unique co-designed HW features of the Efficeon machine and thus cannot be readily applied by either the DVM or the existing DBT on the Efficeon, as demonstrated below.

### 3.3.1 Bytecode Interpretation with LINKPIPE

AccelDroid begins bytecode execution via interpretation in the DBT. Listing 3.3 illustrates the existing bytecode interpreter implementation in the DVM. Different handlers (e.g., OP1_handler and OP2_handler) are stored in a handler table and are used to interpret different bytecodes indexed by the opcode (e.g., OP1 and OP2). At runtime, the bytecode is fetched and executed by jumping to the corresponding handlers based on the opcode. Because an indirect jump to a handler is difficult for an HW branch predictor to predict, the existing interpreter implementation suffers from a high branch mispredictions penalty.

The Efficeon allows the DBT to manage an internal HW queue called LINKPIPE [KN11] to

```
Interpreter:
  fetch a byte code b
  jump to handlerTable[b.opcode]

OP1_handler:
  fetch next bytecode nb
  enqueue handlerTable[nb.opcode] into LINKPIPE
  ...   // interpret OP1
  dequeue LINKPIPE and jump

OP2_handler:
  fetch next bytecode nb
  enqueue handlerTable[nb.opcode] into LINKPIPE
  ...   // interpret OP2
  dequeue LINKPIPE and jump
```

Listing 3.4: DVM Interpreter with LINKPIPE

reduce the number of indirect branch mispredictions. An earlier instruction can enqueue a branch target into the LINKPIPE, and a later instruction can dequeue the branch target from the LINKPIPE and jump to the branch target without any branch prediction. To tolerate pipeline latency, the branch target must be enqueued into the LINKPIPE sufficiently prior to when it is dequeued, which will prevent pipeline stalls.

AccelDroid leverages the HW LINKPIPE feature to reduce indirect branch misprediction in the interpretation of bytecode. We modified the bytecode interpreter to enqueue the handler for the next instruction before the interpretation of the current instruction and to jump to the target dequeued from the LINKPIPE after the interpretation of the current instruction, as shown in Listing 3.4. The interpretation latency is typically more than sufficient to obscure the pipeline latency between the enqueuing and dequeuing of the LINKPIPE.

### 3.3.2 Bytecode Check with SPECULATION

During Dalvik bytecode execution, it is necessary to frequently perform null-pointer and array-bound checks. As an example, Listing 3.5 shows the most frequently executed byte-

```
iget v4, v10, ...          // null-pointer check
iget v5, v10, ...          // null-pointer check
add-int/2addr v4, v5
add-int/2addr v1, v4
add-int/lit8 v0, v0, #int 2 // #02
iget-object v4, v10, ...   // null-pointer check
sub-int v5, v3, v8
aget v4, v4, v5            // null-pointer and array-bounds check
iget-object v5, v10, ...   // null-pointer check
aget v5, v5, v3            // null-pointer and array-bounds check
if-ge v4, v5, 0059
```

Listing 3.5: Bytecode Example

code block in the LOOP benchmark in CaffeineMark. The iget instruction retrieves an instance field from an object, which requires a null-pointer check on the pointer to the object. The aget instruction retrieves an array element from an array object with an array index, which requires a null-pointer check on the pointer to the array object and an array-bound check on the array index. Therefore, overall, we must perform 3 null-pointer checks and 2 array-bound checks for these 11 bytecode instructions (note that redundant-check elimination has already been applied to the code, e.g., the four iget instructions that contain v10 require only one null-pointer check on v10). As a result, a single basic block will be translated into 6 basic blocks on the Efficeon with 5 conditional branches for the checks, as shown in Listing 3.6. The conditional branches for the null-pointer and array-bound checks severely impact the program optimization and instruction scheduling across the branches.

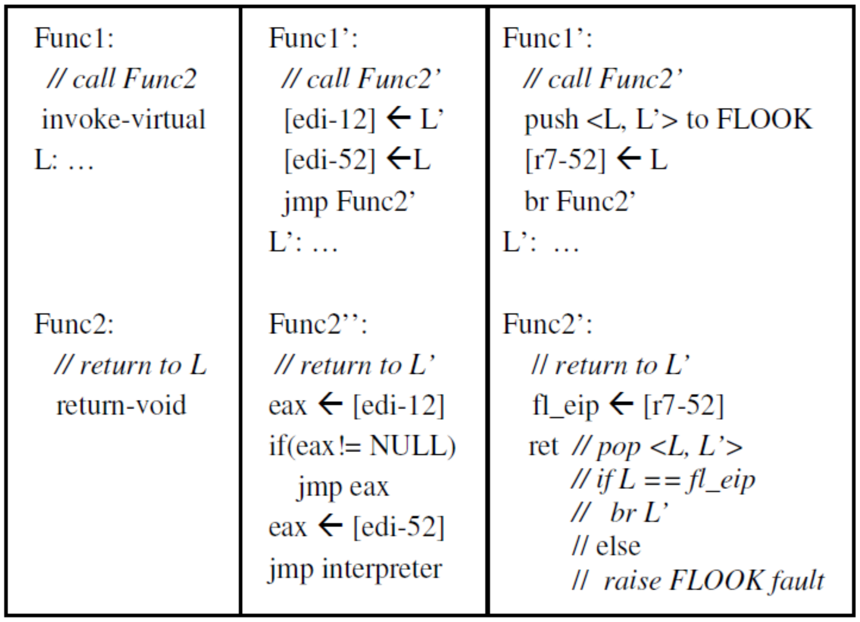
The Efficeon allows the DBT to manage an internal HW SPECULATION feature [NRS+07, PTBC00]. The rarely used conditional branches in a region can be converted into asserts, and speculative optimization and scheduling can be performed across these asserts. In the case of an assert failure, a fault is raised to roll back the execution to the beginning of the optimized region. Then, the non-optimized or less optimized code without the failed assert will be re-executed.

The null-pointer and array-bound checks in bytecode are ideal for asserts because these checks are known to fail only rarely. AccelDroid converts all conditional branches for the null-pointer and array-bound checks into asserts, as shown in Figure 3.7 on page 45. As a result, the bytecode shown in Figure 3.5 on the previous page is translated into a single basic block for aggressive optimization and scheduling across the asserts. The existing DBT in the Efficeon cannot achieve a similar task because after the DVM translates the bytecode into x86 code, all null-pointer and array-bound checks become normal conditional branches, and the high-level information, which consists of extremely biased branches, is lost. Therefore, the DBT will not be able to leverage this information to convert the checks into asserts. Neelakantam et al. [NDZ10] tested a dynamic feedback-based technique to identify biased branches and convert them into asserts, but the reported benefit of this technique is much smaller than that of ours because our approach leverages the high-level bytecode information.

### 3.3.3 Bytecode Call and Return with FLOOK

The DVM's interpreter and translator operate on a stack that is different from the stack that is used in the execution of interpreted and translated bytecode. The x86 stack pointer ESP is reserved for DVM execution in order to maintain the correct x86 state in the presence of interrupts and exceptions, whereas the execution of translated bytecode uses a different register as its stack pointer (e.g., EDI). Because x86 *call* and *ret* instructions implicitly use an ESP register as the stack pointer, the DVM does not translate bytecode function call and return instructions into x86 *call* and *ret* instructions, which would incur additional overhead when swapping the DVM runtime stack pointer and the bytecode stack pointer before and after the calls and returns. Instead, the bytecode function call and return instructions shown in Figure 3.5 (a) are emulated through regular memory accesses and *jmp* instructions, as shown in Figure 3.5 (b).

Specifically, the translated code for the bytecode function call (i.e., invoke-virtual) must store the bytecode return address L into the callee function frame (i.e., address [*edi-52*] ). If the bytecode at the return address L has been translated, then the translated code address L' is also stored in the callee function frame (i.e., address [*edi-12*] ). Otherwise, an invalid NULL address will be stored there. Then, the translated code for the bytecode function return (i.e., return-void) will obtain the translated code address L' from the callee function frame (i.e., address [*edi-12*] ) and jump to it if it is valid (i.e., *eax != NULL*). Otherwise, it jumps to the bytecode interpreter with the bytecode return address L loaded from the callee function frame (i.e., at address [*edi- 52*] ). When the existing DBT in the Efficeon sees the *jmp* instructions in Figure 3.5 (b), it will treat them as regular branches and will not be able to generate internal instructions using the return address stack (RAS) [SAMC98] that is available in the processor to predict the function returns. Our experiment and previous work [KS03] both indicate that this translation of the function call and return instructions to regular branches can significantly degrade performance.



```
Func1:
  // call Func2
  invoke-virtual
L: …

Func2:
  // return to L
  return-void
```

(a) bytecode

```
Func1':
  // call Func2'
  [edi-12] ← L'
  [edi-52] ←L
  jmp Func2'
L': …

Func2'':
  // return to L'
eax ← [edi-12]
if(eax != NULL)
   jmp eax
eax ← [edi-52]
jmp interpreter
```

(b) JITed code

```
Func1':
  // call Func2'
  push <L, L'> to FLOOK
  [r7-52] ← L
  br Func2'
L': …

Func2':
  // return to L'
  fl_eip ← [r7-52]
  ret // pop <L, L'>
      // if L == fl_eip
      //   br L'
      // else
      //   raise FLOOK fault
```

(c) AccelDroid code

Figure 3.5: Translation of Call/Return Instructions with FLOOK

The Efficeon allows the DBT to manage an internal HW FLOOK stack [KN11] for the

```
[basic_block_1]
...
// v10 null check
if (r10 == 0)     // v10 in register r10
    branch to ..  // side exit

[basic_block_2]
...
// v4 null check
if (r4 == 0)
    branch to ...

[basic_block_3]
...
// v5 bound check
if (r5 >= r35)   // bound stays in r35
    branch to ...

[basic_block_4]
...
// v5 null check
if (r5 == 0)
    branch to ...

[basic_block_5]
...
// v3 bound check
if (r3 >= r35)   // bound stays in r35
    branch to ...

basic_block_6:
...
```
Listing 3.6: Checks without SPECULATION

```
[basic_block_1]
...
// v10 null check
raise fault on r10 == 0
...
// v4 null check
raise fault on r4 == 0
...
// v5 bound check
raise fault on r5 >= r35
...
// v5 null check
raise fault on r5 == 0
...
// v3 bound check
raise fault on r3 >= r35
...
```

Listing 3.7: Checks with SPECULATION

efficient translation of function calls and returns, exploiting the HW return address stack (RAS). AccelDroid leverages FLOOK to translate bytecode function calls and returns, as shown in Figure 3.5 (c). Specifically, AccelDroid translates a bytecode function call to *Func2* (i.e., *invoke-virtual*) into a store of the return address L in the callee function frame (i.e., address [*r7-52*] ) and a jump to *Func2'*. In addition, AccelDroid pushes a pair <L, L'> onto the FLOOK stack. Then, AccelDroid translates the bytecode return instruction (i.e., *return-void*) into a load of the bytecode return address (i.e., L, if the function is called from *Func1*) from the function frame (i.e., address [*r7-52*] ) into *fl_eip* followed by a *ret* instruction. The *ret* instruction will pull <L, L'> from the FLOOK stack and compare L with the value in *fl_eip*. In the case of a match, the code will jump to L', which is the translated code corresponding to the bytecode at return address L. In the case of a mismatch, a FLOOK fault is raised, and the DBT runtime will catch the fault and look up the translated code that corresponds to the bytecode return address in *fl_eip* (and may enter interpretation mode if the translated code does not exist). The Efficeon HW implements RAS-based branch prediction [SAMC98] for an *ret* instruction using the FLOOK stack. This scheme is similar to the "dual-address return stack" technique proposed in [KS03].

### 3.3.4 Bytecode Register Promotion

Dalvik bytecode is register-based and can address up to 65,536 virtual (integer or floating-point) registers in each function frame. The large number of available registers allows the DVM to flexibly promote frame memory locations to the architecture registers on the target machine during bytecode translation. In its internal ISA implementation, the Efficeon allows the DBT to manage 128 physical registers - 64 for integers and 64 for floating-point values - which can be used to efficiently promote registers from frame locations. Unfortunately, after the DVM translates the bytecode into x86 code, the majority of the bytecode registers are spilled into bytecode stack memory locations because of the small number of architecture registers available on the x86. Then, in the translation from x86 code to the internal Efficeon code, the DBT must observe all restrictions on x86 binary translation, and it encounters difficulty in promoting the stack memory locations to the Efficeon registers for several reasons.

First, it is expensive and difficult for the DBT to perform accurate runtime binary-level alias analysis among the stack memory locations [GWW+06]. Second, it may be unsafe for the DBT to promote memory to registers because of memory model issues [WW11]. Third, the DBT must preserve the same memory image in the translated execution as in the original x86 binary execution, meaning that the last store to each memory location cannot be removed.

In our AccelDroid implementation, we directly promote the bytecode registers to the Efficeon registers within the optimized region. Across region boundaries, we maintain the live bytecode register values in the frame memory to allow the virtual machine to correctly interface with the DVM. As an example, Listing 3.8 shows the bytecode for the innermost loop of the LOOP benchmark in CaffeineMark. After unrolling the loop once, the DVM sees a single-entry multi-exit region with a total of 118 bytecode register accesses (i.e.,

```
[basic_block_1]
002a: iget v4, v10, LLoopAtom;.FIBCOUNT:I
002c: if-lt v3, v4, 0031       // possible loop exit

[basic_block_2]
0031: iget v4, v10, LLoopAtom;.FIBCOUNT:I
0033: iget v5, v10, LLoopAtom;.dummy:I
0035: add-int/2addr v4, v5
0036: add-int/2addr v1, v4
0037: add-int/lit8 v0, v0, #int 2
0039: iget-object v4, v10, LLoopAtom;.fibs:[I
003b: sub-int v5, v3, v8
003d: aget v4, v4, v5
003f: iget-object v5, v10, LLoopAtom;.fibs:[I
0041: aget v5, v5, v3
0043: if-ge v4, v5, 0059      // possible loop exit

[basic_block_3]
0045: iget-object v4, v10, LLoopAtom;.fibs:[I
0047: sub-int v5, v3, v8
0049: aget v4, v4, v5
004b: iget-object v5, v10, LLoopAtom;.fibs:[I
004d: sub-int v6, v3, v8
004f: iget-object v7, v10, LLoopAtom;.fibs:[I
0051: aget v7, v7, v3
0053: aput v7, v5, v6
0055: iget-object v5, v10, LLoopAtom;.fibs:[I
0057: aput v4, v5, v3
0059: add-int/lit8 v3, v3, #int 1
005b: goto 002a               // branch to 002a
```
Listing 3.8: Register Promotion Example

v3, v4, etc.). Without AccelDroid register promotion, the translated internal Efficeon code contains a total of 281 instructions, with 57 frame memory location accesses. With AccelDroid register promotion, the translated code contains only 121 instructions, with 24 frame memory location accesses.

### 3.3.5 Optimizing 64-Bit Loads and Stores

In our performance analysis, we noticed an increased number of misaligned exceptions [LWH11b] in the bytecode instructions aget-wide and aput-wide, which load and store 64-bit values from and to memory, respectively. These exceptions do not occur if the bytecode is translated into x86-ISA and then retranslated into the internal ISA by the DBT. The DBT prevents unaligned 64-bit load/store operations by leveraging the dynamic profile information [CHH+98a, CHH+98b] at the DBT runtime. Because AccelDroid is also translating at the DBT level, we added one optimization to check whether the aput-wide and aget-wide operations contained in the profile data being compiled have previously caused an unaligned load/store exception. If the profile indicates that an instruction caused an exception, then instead of only one 64-bit load (store), two 32-bit loads (stores) and one shift operation are generated to load (store) the 64-bit value. By applying this optimization, we eliminate the unaligned exceptions caused by the aget-wide and aput-wide bytecode instructions.

## 3.4 Experiments

We tested AccelDroid using an x86-based Android system, version 2.3, codenamed Gingerbread, which was the most recent official Android system to be installed on an Intel smartphone when the experiment was performed. We implemented a new AccelDroid PCI device driver in the Android Linux Kernel and modified the DVM to leverage AccelDroid for bytecode acceleration, as discussed in Section 3.2.2 on page 35.

We modified the existing DBT (which is called CMS, or Code Morphing Software [DGB+03]) in the Efficeon machine to implement a virtual AccelDroid PCI device, which supports direct Dalvik bytecode execution by translating bytecode into internal Efficeon code.

### 3.4.1 Programmability

Because our acceleration interface uses the standard hardware I/O interface and is designed to accelerate bytecode execution on the CPU pipeline and not on a separate hardware accelerator, it considerably simplifies the VM implementation in order to enable bytecode acceleration, thereby avoiding the complicated handling of synchronization and communication between the CPU and the accelerator. In addition, the accelerated bytecode execution directly accesses and updates the CPU architecture states without the need for explicit data communication between the x86 ISA code execution and the accelerated bytecode execution. Furthermore, our design avoids many common issues of dedicated hardware accelerators, such as long access latency, large die area, data marshaling and unmarshaling, and high costs.

We implemented an acceleration driver and plugged it into the Android OS. We modified the DVM to accelerate Dalvik bytecode execution using the acceleration driver. Overall, we added $\sim$150 lines of code for the acceleration driver and modified $\sim$30 lines of code in the DVM.

### 3.4.2 Porting the Dalvik Runtime into the Efficeon DBT

For performance reasons, only the hand-optimized assembly version of the Dalvik interpreter is integrated with the JIT compiler in the Gingerbread version of the Android OS. This distribution also contains a portable version of the interpreter, written in C, which is commonly used as a reference implementation and for debugging. We ported the C interpreter to our AccelDroid runtime and integrated it with the DBT profiling and JIT compilation/dispatch. We also ported the trace selection module from the DVM JIT compiler to our accelerator. The DVM JIT compiler is a trace-based compiler [GPF06a, GF06], and the trace selection module is needed for the selection of hot bytecode regions for translation because the original Efficeon DBT is only capable of selecting hot x86 regions. Whenever a new bytecode trace is formed, the bytecode-aware DBT JIT compiler translates it into the DBT low-level IR and performs all of the bytecode-specific optimizations in the DBT, as discussed in Section 3.3 on page 39. After the bytecode-specific optimizations are performed, the low-level IR is passed to the DBT JIT backend, which will schedule, assemble and install the translation into the internal DBT code cache.

### 3.4.3 ISA Compatibility and Transparency

Our acceleration interface is provided by an OS acceleration driver via privileged architecture I/O instructions (e.g., x86 in/out instructions) without the need to extend the x86 (architectural) ISA. Bytecode-specific information is passed through the accelerator interface to the DBT for bytecode-specific translation and optimization. In this manner, 1) we enable the microarchitectural extension of the internal ISA and the CPU pipeline design to transparently accelerate the general-purpose managed code execution without any change in the architectural ISA, and 2) we become able to run the bytecode code transparently and efficiently through the embedded DVM runtime, closely coupled with the CPU pipeline. We make no assumptions about or changes to the OS code, except for the addition of a new OS kernel accelerator driver.

### 3.4.4 CaffeineMark 3.0 Benchmark

We collected performance data using the CaffeineMark 3.0 benchmark, which is a commonly used benchmark for measuring the performance of Android systems and has been widely used to assess DVM performance on more than 600 Android phones. CaffeineMark 3.0 contains a series of tests that are designed to evaluate various modules of the DVM runtime. The following provides a brief description of each test:

- **Sieve.** The classic sieve of Eratosthenes identifies prime numbers. This test primarily evaluates integer division operations.
- **Loop.** The loop test uses sorting and sequence generation to measure compiler optimization of loops.
- **Logic.** This test evaluates the speed with which the virtual machine executes decision-making instructions and branch execution.
- **Method.** The Method test executes recursive function calls to observe how well the VM handles method calls.
- **Float.** This test simulates the 3D rotation of objects around a point.
- **String.** This test involves basic string operations. This test also evaluates garbage collection performance. More than 90% of the time is spent on native execution, and only approximately 10% is spent on bytecode execution.

CaffeineMark produces, for each test, a score that represents the number of Java instructions executed per second. In our experiments, the final score included the just-in-time compilation overhead in the AccelDroid runtime.

### 3.4.5 Overall Performance Benefits

Figure 3.6 on the following page shows the overall performance benefit achieved using AccelDroid measured in terms of speedup with respect to the baseline. The left-most bar represents the baseline performance on the Efficeon machine when running through the existing Android DVM and Efficeon DBT. The right-most bar represents the performance achieved when running through AccelDroid. Overall, AccelDroid achieves an 83% speedup on the Efficeon over the baseline. This speedup is 5% higher than that reported in our previous work [WCW13] because of the new optimization technique applied in the 64-bit load and store bytecodes, as described in Section 3.3.5 on page 48. The existing DVM implementation on the Android platform translates bytecode into x86 code only in basic block units. The underlying DBT can translate x86 code in large "*superblock*" regions across these basic blocks [HMC+93]. AccelDroid directly translates large regions of bytecode into the internal Efficeon code. Considering that a future Android DVM may implement large-region-level bytecode translations, we modified the DVM to also address the same bytecode regions as AccelDroid for a fair comparison. The corresponding performance is represented by the middle bar. Even with large-region translation in the DVM (i.e., DVM_LR/DBT), AccelDroid can still improve the performance by 68%.

Figure 3.7 on page 54 shows the performance benefit of AccelDroid on some popular algorithms. Acceldroid achieves great speedups for the two heavily recursive applications Ackerman and Fibo(Fibonacci). For the loop intensive Matrix Multiplication (Matrix Mult) algorithm, AccelDroid achieves a 2x speedup. Overall, AccelDroid achieve 81% speedup for this set of applications.

To compare the performance of AccelDroid on the HW/SW co-designed Efficeon machine

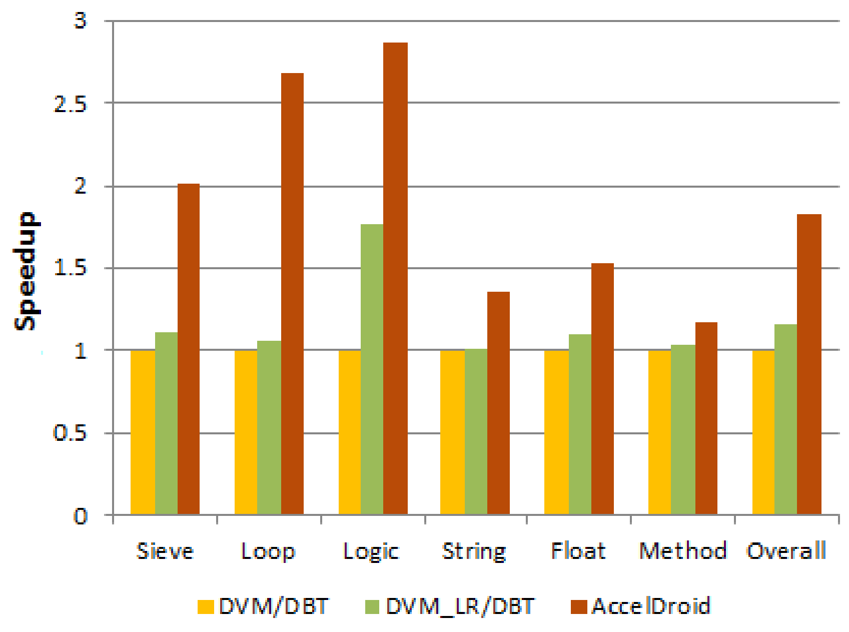


Figure 3.6: Speedup on CaffeineMark

with its performance on normal x86 processors without HW/SW co-design, we also measured its performance on two normal x86 processors: an early version of the Intel Atom processor [Int08], which can be found in many netbooks and performs similarly to the Efficeon processor, and a recent Intel Ivy Bridge (IVB) processor [Int12]. Both the Atom N270 and the Efficeon are in-order processors and contain significantly fewer transistors than does the high-end Ivy Bridge system, which is a superscalar processor for high-end PCs and operates at higher power levels (77 watts). Table 3.1 on the next page shows the specifications of these three processors.

Because the different processors have different power/performance budgets, for a fair comparison we used the SPEC CPU2000 benchmarking software to calibrate the performances of the different processors, as shown in Figure 3.8 on page 55. The figure shows that the Efficeon exhibits a performance that is similar to that of the Atom and approximately 1/8 the performance of the IVB. Note that SPEC programs are not bytecode programs, and

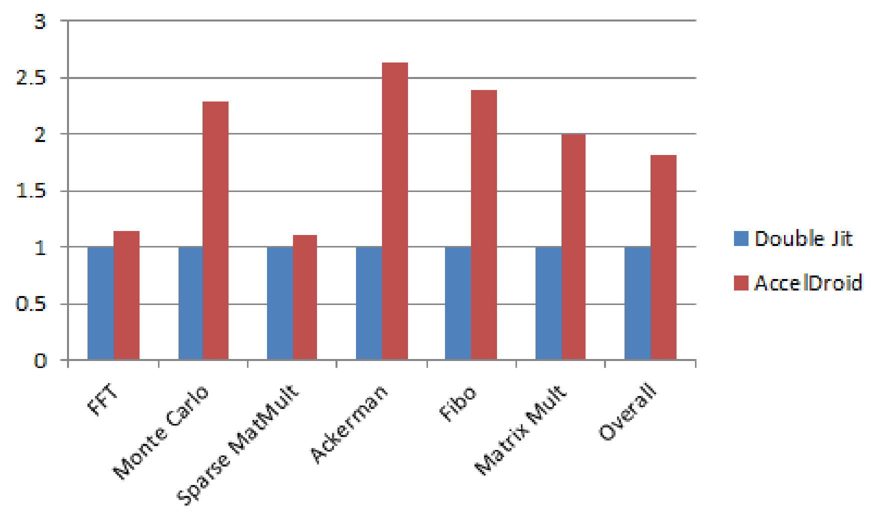


Figure 3.7: Acceldroid speedup on some popular algorithms

| Processor | Atom N270 | Efficeon | IVB i7-3770K |
|---|---|---|---|
| **Frequency** | 1.6 GHz | 1.6 GHz | 3.5 GHz |
| **Cores** | 1 | 1 | 4 |
| **Launch Date** | 2008 | 2004 | 2012 |
| **Transistors** | 45 million | 85 million | 1.6 billion |
| **Process Scale** | 45 nm | 90 nm | 22 nm 3-D |
| **Thermal Design Power** | 2.5 watts | 4.3 watts | 77 watts |
| **Die Area** | 26 mm$^2$ | 65 mm$^2$ | 160 mm$^2$ |
| **L1 D-Cache** | 6-way 24 KB | 8-way 64 KB | 8-way 4x32 KB |
| **L1 I-Cache** | 8-way 32 KB | 4-way 128 KB | 8-way 4x32 KB |
| **L2 Cache** | 8-way 512 KB | 4-way 1 MB | 8-way 4x256 KB |
| **L3 Cache** | None | None | 16-way 8 MB |
| **Memory** | 1 GB | 1 GB | 8 GB |

Table 3.1: Processor Specifications

thus they cannot benefit from AccelDroid.

Figure 3.9 on the next page shows the CaffeineMark performance on the various processors. With AccelDroid, the Efficeon can achieve approximately twice the performance of the Atom and half the performance of the IVB. If SPEC CPU2000 were to be run as Dalvik bytecode, we would expect that AccelDroid would greatly improve its performance on the Efficeon machine, as well.

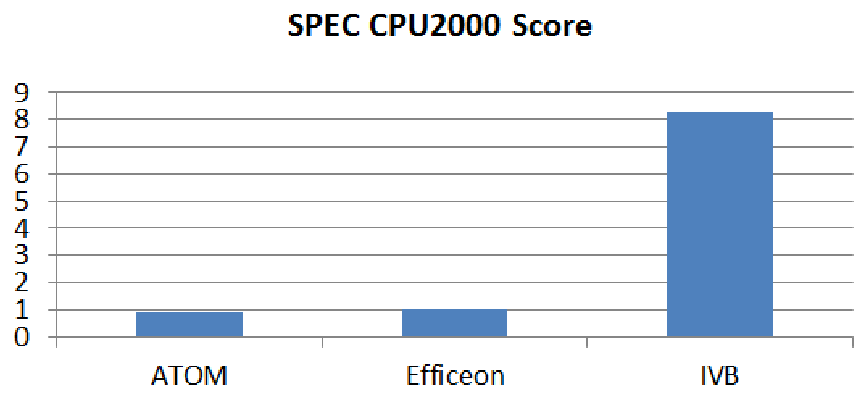


Figure 3.8: SPEC CPU 2000 Performance Comparison

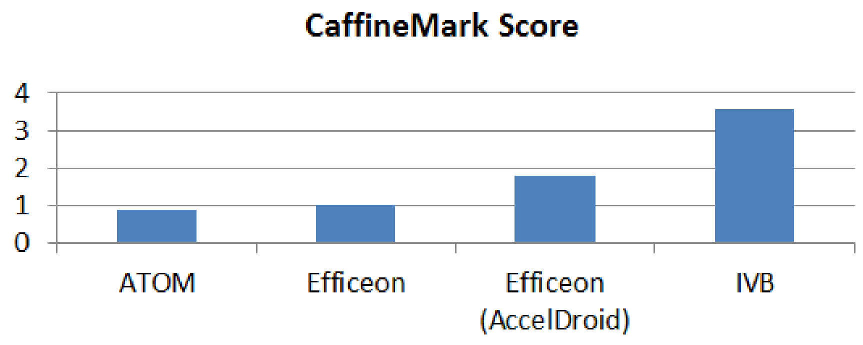


Figure 3.9: CaffeineMark Performance Comparison

### 3.4.6 Bytecode-Specific Optimization Benefits

To illustrate the benefits of the bytecode-specific optimizations in detail, we assessed the individual performance benefits, as shown in Figure 3.10 on the following page. The first bar (AccelDroid) represents the overall AccelDroid speedup. The remaining bars represent the performances achieved by disabling the optimizations one by one. The second bar (NO_REG_PMT) represents the AccelDroid speedup without the register promotion optimization (see Section 3.3.4 on page 46).

The third bar (NO_REG_PMT + NO_FLOOK) represents the speedup achieved when the FLOOK optimization (see Section 3.3.3 on page 42) was also disabled. The final bar (NO_REG_PMT + NO_FLOOK + NO_SPEC) represents the speedup obtained upon further disabling the SPECULATION optimization (see Section 3.3.2 on page 40). Overall, more than 50% of the 83% speedup originates from these three bytecode-specific optimizations. The remaining speedup originates from overcoming other inefficiencies in the execution through DVM/DBT, such as the slower code cache warm-up and the double translation overhead, although these benefits cannot be easily separated out individually. The frame register promotion considerably improves the performance on Loop, Logic and Float, which use many registers for their computations. The FLOOK optimization is especially beneficial for Method because of the frequent function calls and returns in this benchmark. The SPECULATION optimization is most beneficial to Sieve and Loop, which contain numerous null-pointer and array-bound checks. String does not gain much benefit from these optimizations because the majority of its execution is performed in the native x86 library code (invoked through JNI calls), and only the small percentage that is performed through bytecode execution benefits from the bytecode-specific optimizations.

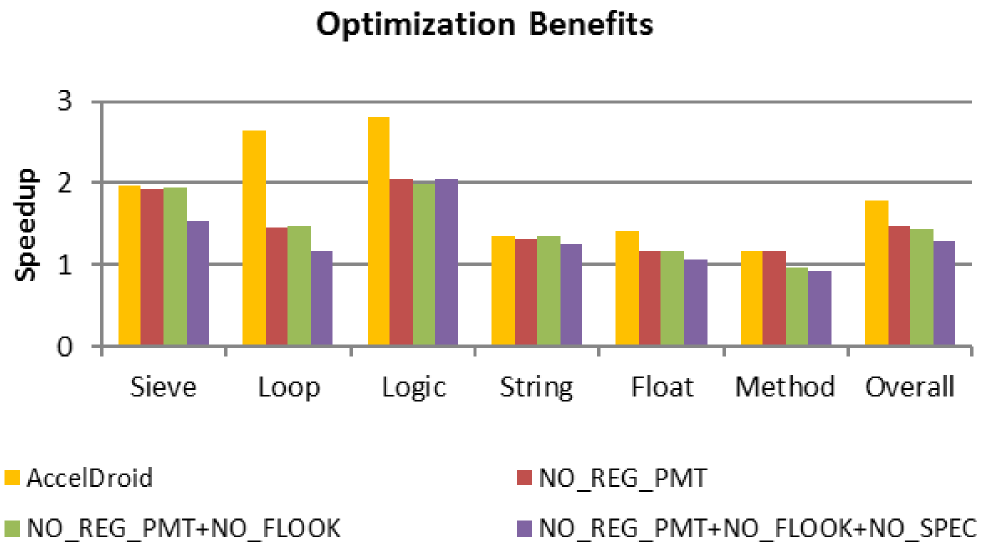


Figure 3.10: Optimization Benefits

### 3.4.7 Direct Interpretation and LINKPIPE Benefits

One common concern regarding dynamic languages, especially on mobile platforms, is the cold code execution performance. When an application first starts up, it is run as cold code, which runs through slow interpretation. The cold code performance affects much of the user experience because most interactive operations are performed in this manner. CaffeineMark contains mostly hot code, which is ideal for measuring the performance in bytecode translation but incurs little bytecode interpretation overhead.

To measure the interpretation performance, we used interpretation only (i.e., translations were disabled) in the DVM, the DBT and AccelDroid; the resulting performances are depicted in Figure 3.11. Overall, AccelDroid achieved a speedup by more than a factor of 20 compared with the DVM/DBT implementations. This is reasonable because interpretation typically results in slowing the program execution by more than a factor of 20 compared with native execution.

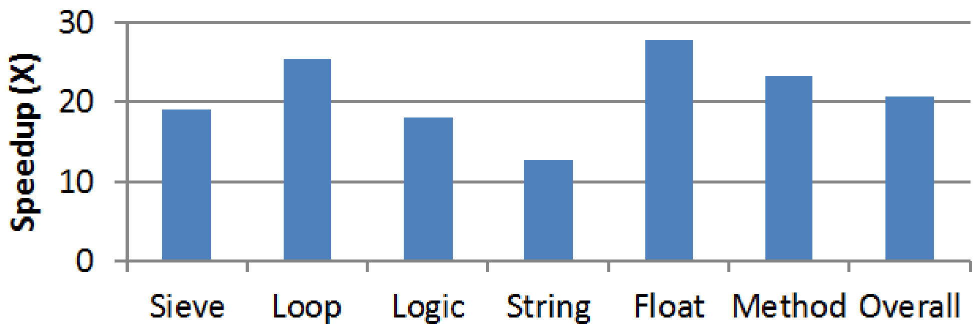


Figure 3.11: Interpretation Performance

Figure 3.12 on the next page shows the benefits of the LINKPIPE optimization in bytecode interpretation. The left-hand bar represents the performance of AccelDroid in interpretation-only mode without the use of LINKPIPE, and the right-hand bar represents the performance with LINKPIPE. Overall, through LINKPIPE optimization, we obtained performance improvements of approximately 25% by avoiding the indirect branch miss overhead that typically arises during the interpretation process.

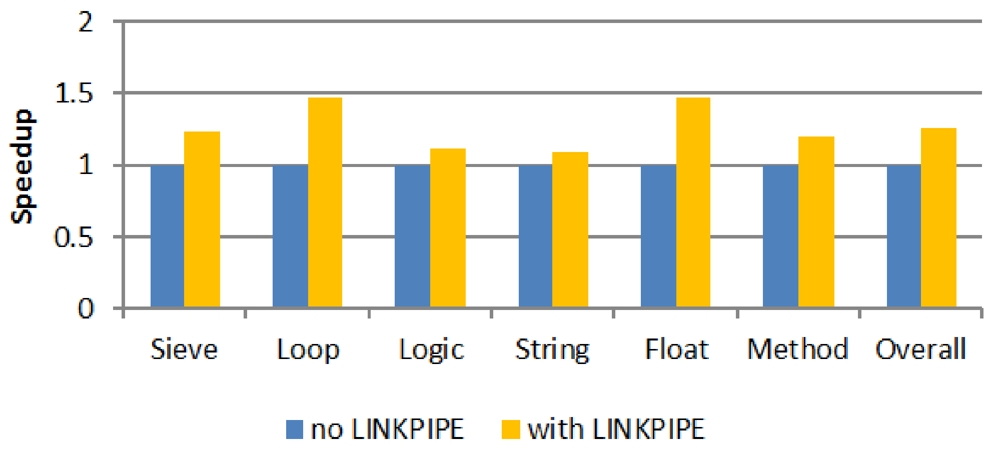


Figure 3.12: LINKPIPE Benefit

### 3.4.8 Energy Benefits

Brooks et al. [BBS+00] showed that the Power-Delay Product (PDP) is an appropriate formula for determining the energy efficiency of low-power, portable systems, in which the battery life is the primary concern. Our experience indicates that the PDP of a program is closely correlated with the number of dynamic instructions executed on the Efficeon machine; therefore, we used the dynamic instruction count to model the energy efficiency. Figure 3.13 on the following page shows the energy savings of AccelDroid with respect to the baseline measured in terms of the dynamic instruction count. On average, AccelDroid exhibits an energy savings of 41%, as measured on the CaffeineMark benchmark suite. String demonstrates the lowest energy savings because the majority of its execution is performed in the native x86 library code (invoked through JNI calls) instead of bytecode.

The energy analysis shown in Figure 3.13 on the next page is performed after the CaffeineMark core tests have warmed up from JIT compilation for 1 second. We also executed each core benchmark for a fixed number of iterations such that both AccelDroid and DVM_LR/DBT ran the same amount of bytecode including JIT warmup. Figure 3.14 shows that the energy savings of AccelDroid including JIT warmup are 58%, an increase of 17% in energy savings compared to the stable execution after warmup. These increased energy

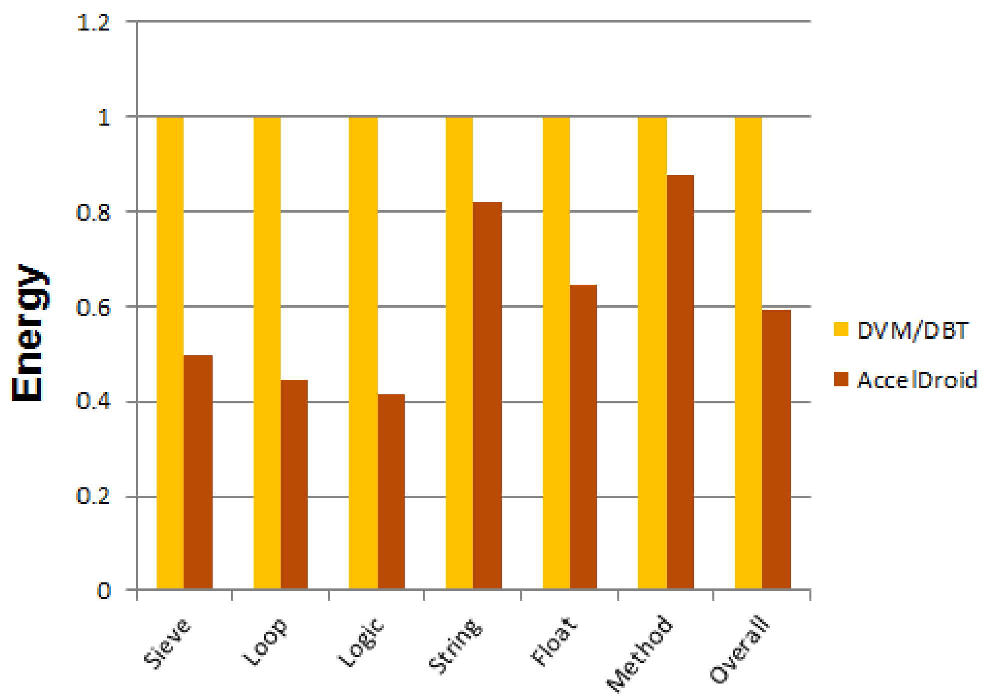


Figure 3.13: Energy Savings without warmup overhead

savings are due to the fact that AccelDroid is performing direct translation of bytecode into the internal VLIW ISA, which incurs less JIT compilation overhead compared to the double translation of bytecode: first to x86, then to the internal VLIW ISA. AccelDroid also leverages the Efficeon's hardware support to reduce the program startup overhead, such as runtime profiling and fast interpretation suports. Figure 3.15 on page 61 shows some performance counters for the energy analysis of Figure 3.14 on the next page. AccelDroid significantly reduces the amount of translation and interpretation cycles due to the direct translation of bytecode into the VLIW ISA. We noticed a 44% increase of interpretation cycles for String due to garbage collection(GC) execution in the DVM, which executes as native x86 code. The overall cycle reduction for String is lower compared to the other benchmarks, due to GC, but AccelDroid still produces a 40% execution speedup. We also observed lower Load Miss and Write Queue (WQ) Full exceptions in AccelDroid due to the register promotion optimization. Lastly, AccelDroid reduces the translation code size, since it leverages both the bytecode semantics and the hardware speculation support we described in Section 3.3.2 on page 40, resulting in reduced instruction cache misses (Icache miss bubbles) during execution.

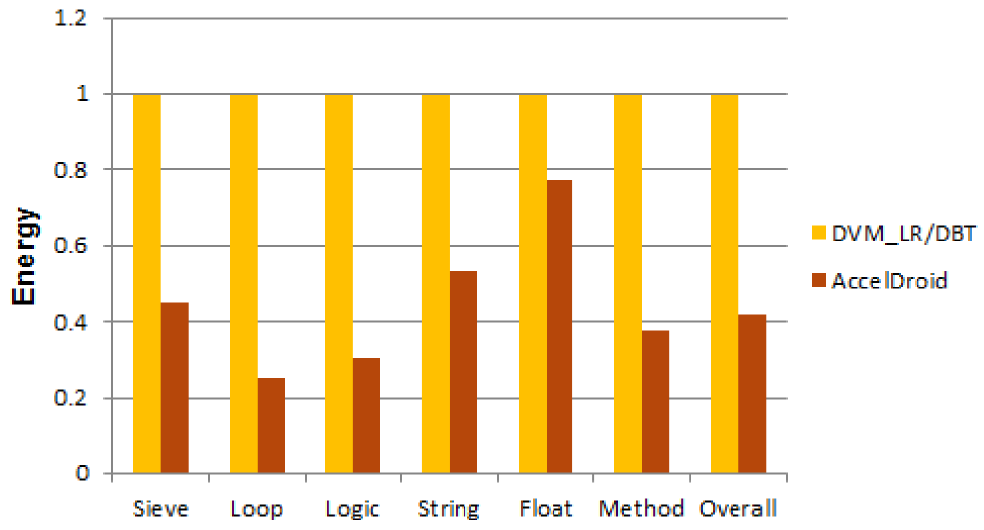


Figure 3.14: Energy Savings including JIT warmup

## 3.5 Conclusions

In this Chapter, we present AccelDroid, an HW/SW co-designed system that translates Android Dalvik bytecode into the code of the internal customized ISA implementation, thereby achieving more efficient execution. Our experimental results show that AccelDroid can significantly improve the bytecode execution performance on HW/SW co-designed systems such as the Transmeta Efficeon.

AccelDroid offers several advantages that can be used in innovative mobile processor designs. This work can be expanded in three directions. First, Java/Dalvik bytecode is only one of the dynamic languages that are used in application development for mobile systems. AccelDroid can be extended to accelerate other dynamic languages such as PHP, Python, and JavaScript, by using a common application programming interface (API) that each dynamic language runtime could use to interface with the HW/SW co-designed virtual

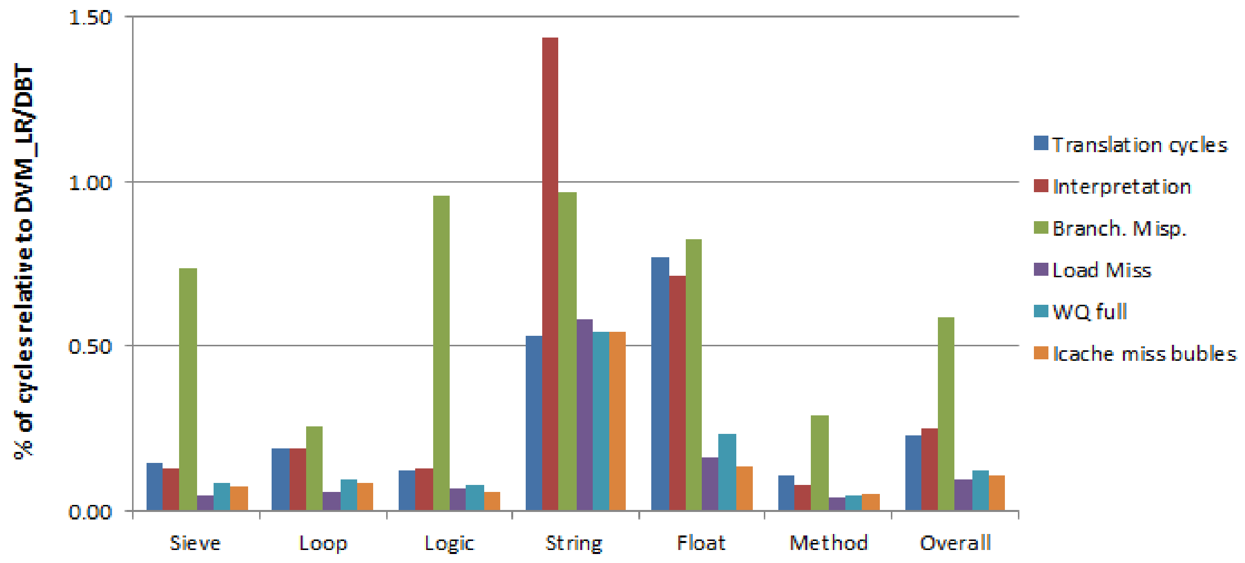


Figure 3.15: Performance Counters for the Energy Analysis of Figure 3.14 on the preceding page

machine, so that a single internal compiler could accelerate all major dynamic languages [Wu13].

Second, AccelDroid enables customized internal ISA implementations and microarchitectural extensions in the underlying hardware for efficient bytecode execution without affecting the ISA backward compatibility. In this work, we primarily leveraged the existing co-designed HW features in the Efficeon, which was originally designed to support translation for x86 instructions. We believe that new co-designed HW features specifically designed for accelerating dynamic languages can further increase the energy/performance efficiency of AccelDroid.

Third, we believe our framework provides the groundwork for future research on innovative co-designed security schemes to leverage our acceleration interface to further reduce the overhead of strong security policies enforcement, during program execution, with little or no increase in energy consumption on mobile devices.

# Chapter 4

# Evaluating Bytecode Performance and Energy Efficiency on Processor Simulators

*"Simplicity is prerequisite for reliability."*

Edsger Dijkstra

In this chapter we present a novel technique to evaluate the performance of bytecode execution based on the acceleration interface used to implement AccelDroid, introduced in the previous chapter, which shows that our transparent and efficient acceleration interface has applications other than those of merely achieving higher performance-per-watt. The simulation control technique presented in this chapter is used to create simulation checkpoints for the workloads used to evaluate the implementation of AccelDroid on a Out-of-Order processor using a cycle accurate simulator, which is the subject of  Chapter 5 on page 85.

## 4.1 Difficulties for Bytecode Energy/Performance Evaluation on Processor Simulators

Driven by a motivation to increase developer productivity, managed runtime environments (Virtual Machines) emerged in the mid-1990s and provided a generalized way to support the sophisticated features of modern high level languages, such as object-orientation, security, robustness (automatic memory management), networking, and performance. A managed runtime environment is a platform that abstracts away the specifics of the operating system and the computer architecture running beneath it. To do so, it executes a platform independent ISA that has not been designed with direct hardware implementation in mind. The Java [LY99], Microsoft Common Language Runtime (CLR) [MWG00], JavaScript [ecm], and PHP [Php] are the most popular virtual machines in production systems today.

With the massive shift to mobile technologies, high level languages that run on top of a Virtual Machine have become a very popular method of implementing the applications that execute on the software stack of these mobile systems, offering a foundation for quick mobile software development and deployment on different hardware architectures.

As these virtual machines execute on top of different hardware platforms, it has become increasingly challenging to evaluate the energy/performance efficiency of bytecode applications on different architecture designs, as the bytecode is translated into different forms of intermediate representations and binaries, across the software stack, before it is executed by the native architectural ISA. Since different processors may support different architectural ISAs, or support the same architectural ISA but have different implementation ISAs (I-ISA) - the actual ISAs executed on the CPU pipeline - it becomes challenging to evaluate the performance and energy efficiency of bytecode applications on these processors.

An essential task in designing a new computer architecture is the careful examination of different design options. Different architecture designs are usually compared by running the same workload on each architecture and comparing the performance. Architectural simulation is an indispensable tool for architecture design exploration, as it enables the designers to predict the performance, energy efficiency and reliability of a new architecture before a working prototype of the processor is available. Architectural design space exploration becomes more difficult when different binaries must be used to represent the same program. For example, the same source program can be statically compiled to a 32 bit native binary of CPU X, or to a 64 bit binary of a CPU Y, as illustrated in Figure 4.1 on the following page.

Given that different binaries are used for the same program source, it is challenging to evaluate the performance-per-watt of the same region of application code on different architecture simulators, unless there are indicators for the beginning and ending of the code regions. This could be achieved by executing the program from beginning to the end, but that's not feasible to perform in an architecture simulator due to the long execution time. So the performance is traditionally only collected on hot regions of code. This is usually achieved by either using special escape instructions, special opcodes, or rarely used instructions in order to delimit the regions of code that are representative of the current program behavior. However, it is not always safe to use rarely used instructions, since they may still be used by an old library, or an OS service, which may prevent their use for performance analysis, or produce misleading results if not carefully inspected.

The same complication exists when the application is compiled into an architecture independent Intermediate Representation (IR), for example the bytecode, for execution in a Virtual Machine (VM), as Figure 4.5 on page 76 illustrates. In this case, the same bytecode is dynamically compiled, by a Just-in-Time (JIT) compiler, into the native code of the architecture ISA where the VM is running. Given that the same bytecode is compiled into different ISAs (CPU X or CPU Y in Figure 4.5 on page 76), a mechanism must exist in order to precisely evaluate the performance of a bytecode region of interest.

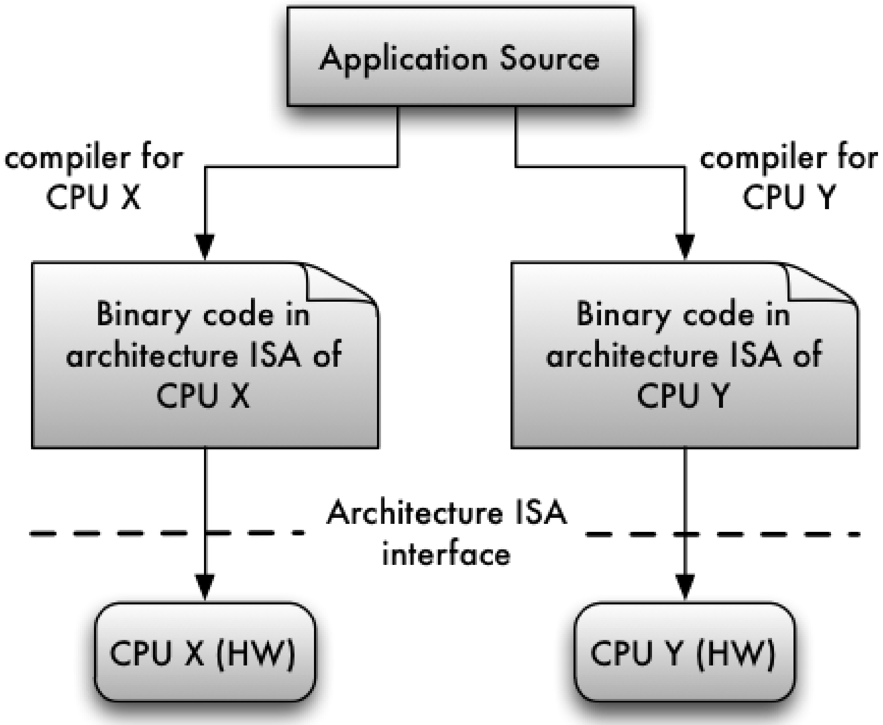

Application Source
compiler for CPU X
compiler for CPU Y
Binary code in architecture ISA of CPU X
Binary code in architecture ISA of CPU Y
Architecture ISA interface
CPU X (HW)
CPU Y (HW)

Figure 4.1: Application source being compiled into different binaries for execution

When multiple binaries are used to evaluate a program, one approach is to create a separate set of simulation points (also called simulation checkpoints) for each workload used for energy and performance evaluation. Selecting a single set of simulation checkpoints to represent a program execution across multiple binaries/IRs is a challenging task for the simulator developer, since it may require modifications to the tools and to the application runtime. This problem domain is the focus of this chapter. To tackle this problem, we present a novel technique that simplifies this process by reducing the implementation effort necessary to enable precise energy and performance evaluation of bytecode applications on different processors, expediting the processor design exploration, and reducing time to market and costs. To the best of our knowledge, our scheme is the first solution to this problem that minimizes the implementation effort to enable precise performance evaluation of different architecture design implementations.

This chapter makes the following contributions to the state of art of performance evaluation of bytecode applications on CPU simulators, which we will describe in detail in the next sections:

- It presents the design of a novel scheme to allow the performance evaluation of bytecode applications in architecture simulators by providing APIs to delimit a region of bytecode over which energy/performance metrics are to be gathered in the simulator. Our scheme does not require changes to the compiler/JIT in the virtual machine, nor the use of unsafe escape/rarely used/new instructions;
- It shows evidence of the value of our scheme by providing an implementation on top of a research state-of-art x86 processor simulator to evaluate the performance of bytecode applications running on the Android Dalvik Virtual Machine, with low execution overhead; Our technique also enables the generation of simulation checkpoints for easy and fast processor designs evaluation;
- It demonstrates that our technique reduces the implementation effort to modify the application software stack and the cycle accurate architecture simulator to control the bytecode execution, allowing computer architects and system software developers to have an efficient means for estimating the impact of various design options on the overall machine;

We claim that our efficient instrumentation control scheme can also be applied for the purposes of debugging, profiling, and performance monitoring of an application running on a real machine.

## 4.2 Motivation for a Virtualized Interface for Simulation Control

In this section we show the technical challenges of benchmarking bytecode applications on top of architecture simulators for different processor architectures.

### 4.2.1 Cross-platform Benchmarking

Cross-platform energy-performance analysis is commonly achieved by modifying the application source and adding special compilation directives to delimit regions of interest. The compiler for each architecture is modified to produce special instructions / opcodes when the compilation directives are parsed; for example, the IA-32 instruction *aaa* (adjust after addition) instruction. If the application is written using a language that requires a managed runtime for execution, the portable intermediate representation also needs to be extended with new instructions, which will also require modifications in the VM runtime (interpreter and JIT compiler). Finally, each architecture simulator has to be modified, adding special handling for the new instructions/opcodes, so that it can control the simulation execution.

Current state-of-the-art simulators use special escape instructions in the binaries/IR to mark the beginning and the end of a given region of interest for performance evaluation. The escape instructions have a major drawback, since all of the interpreters and dynamic/static compilers must be modified in order to support them. Also, the simulator must be modified in order to handle the special escape instructions, which complicates simulator design and implementation. Our technique, in contrast, requires only that the simulator control unit watches the special bytecode marker (implemented using a simulation control counter (*scc*), as we will see), controlling the simulation execution according to the values

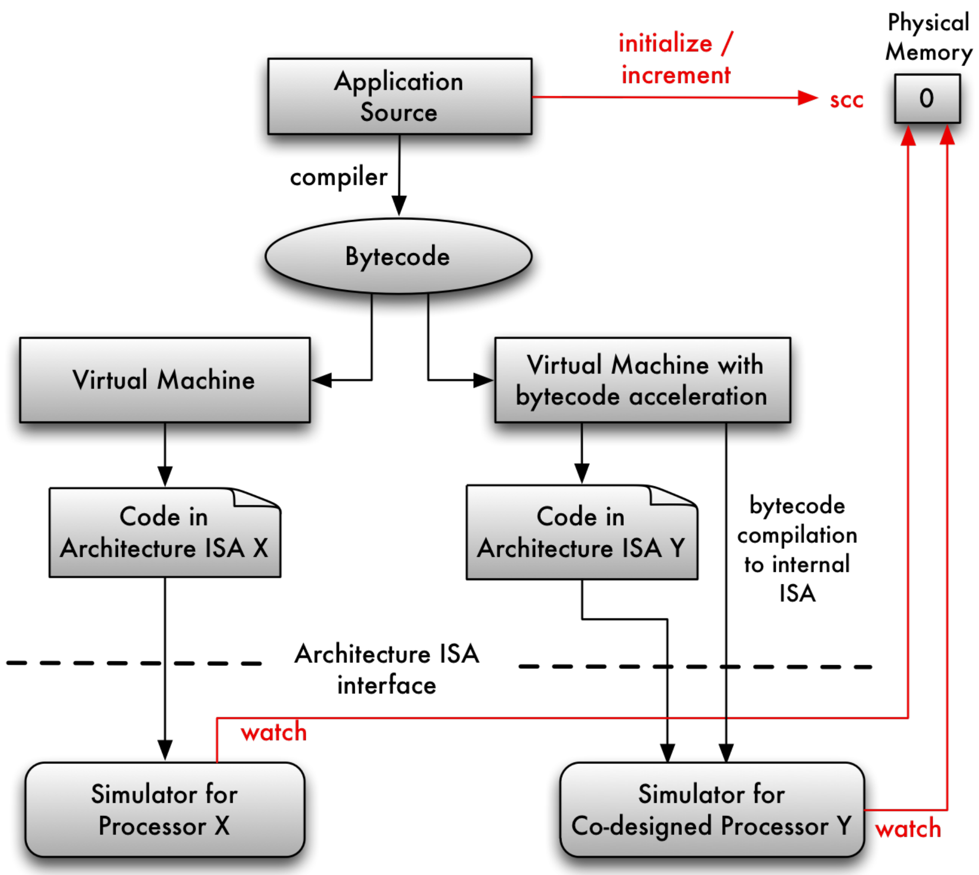

Application Source
initialize / increment
scc
Physical Memory
0
compiler
Bytecode
Virtual Machine
Virtual Machine with bytecode acceleration
Code in Architecture ISA X
Code in Architecture ISA Y
bytecode compilation to internal ISA
Architecture ISA interface
watch
Simulator for Processor X
Simulator for Co-designed Processor Y
watch

Figure 4.2: Simulation Control Interface on A HS/SW co-designed System

of the marker during the program execution. This results in fast simulator development, as the developer does not need to worry about ISA extensions. Furthermore, our technique does not require any modifications in the software stack, in the layers between the application source and the underlining architecture simulator, thus reducing the implementation effort and consequently accelerating the architecture design exploration [PHC03a].

### 4.2.2 Controlling the Simulation on a Co-designed Processor

To illustrate that the use of special escape instructions to control the energy-performance simulation analysis is a large engineering effort, we demonstrate the steps required to implement it in a state-of-art architecture simulator for a mobile processor.

In a VM with a HW/SW co-designed bytecode accelerator, such as the one we presented in [WCW13], adding a special bytecode to delimit a region of interest, for means of energy/performance evaluation, is a complex infra-structure modification, since significant modifications are needed to the bytecode accelerator interface, to the underlying bytecode runtime, and to the DBT virtual accelerator, in order to support the new bytecode. The scheme we propose in this chapter simplifies the modifications to an existing architecture simulator for a co-designed processor, since it reduces the implementation and debugging efforts of the non-trivial task of adding a new bytecode instruction. As shown in Figure 4.2 on the previous page, a new simulation control counter (scc) is defined and initialized to 0 (zero) at application level. The application used for performance evaluation is modified, adding calls to increment the scc at certain points of interest. The simulator engine is extended to watch the values of scc, and stop the application execution based on the values assumed by the counter, when hardware counters metrics, used to evaluate performance and energy consumption, are dumped. Our scheme requires no modifications to the application compiler, VM (and its components), Architecture ISA, DBT, or to the internal Implementation ISA (I-ISA).

## 4.3 Benchmarking Using Special Markers

In this section we present an overview of our scheme to control the simulation engine to perform energy/performance analysis of bytecode applications on different architectures. We present a new API to create and manipulate the simulation control counter (scc or bytecode marker), at program source level. We then show the modifications in the OS, virtual machine environment, and simulator software necessary to implement the bytecode marker.

```
public class SCC {

  // native API to control the
  // scc: simulation control conunter
  public native void init(int value);
  public native int get();
  public native void increment();

  public SCC(int initValue) {
    this.init(initValue);
  }

  public SCC() {
    this(0);
  }

  static{
    System.loadLibrary(``simulatorNativeAPI'');
  }
}
```

Listing 4.1: Simulation Control Class (SCC) - main API to control scc updates

### 4.3.1 API for Bytecode Instrumentation

Listing 4.1 shows the Simulation Control Class (SCC) definition. This class provides the main services needed to instrument bytecode applications written in the Java programming language. When a new instance of the class is created, the constructor will call the *init* method, which will use the java native interface to initialize the simulation control counter (*scc*) with the initial value provided.

### 4.3.2 OS Simulation Control Driver

We implement the OS simulation driver as shown in the flow chart in Figure 4.4 on page 73. The simulation driver interacts with the virtualized device driver for simulation control, implemented by the simulation application, across the simulation interface with privileged architecture I/O instructions. During the OS boot time, the simulation interface is probed and the simulation driver is loaded and initialized accordingly. The simulation driver initialization reserves MMIO address space from the OS and sends it to the virtualized device driver through the simulation interface. The MMIO address space is used by the virtualized device driver to set up the simulation execution control unit, which can be used to stop the bytecode execution when the virtualized simulation counter reaches a certain value of interest. The bytecode application will increment the virtualized counter by calling the method *increment* in our simulation interface, as shown in Listing 4.1.

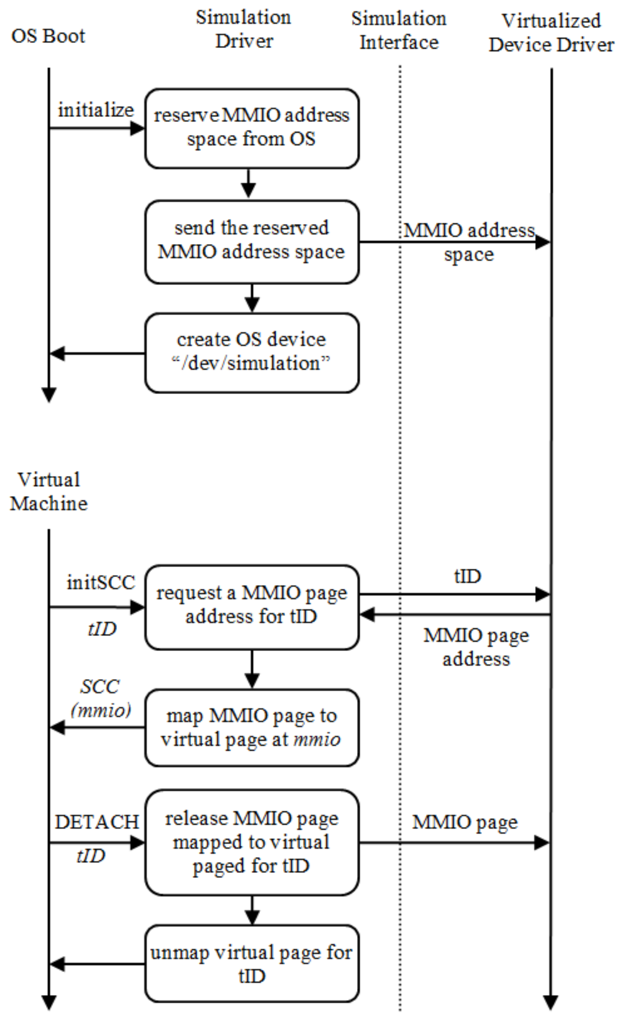

OS Boot
Simulation Driver
Simulation Interface
Virtualized Device Driver
initialize
reserve MMIO address space from OS
send the reserved MMIO address space
MMIO address space
create OS device "/dev/simulation"
Virtual Machine
initSCC
tID
request a MMIO page address for tID
tID
MMIO page address
SCC (mmio)
map MMIO page to virtual page at mmio
DETACH
tID
release MMIO page mapped to virtual paged for tID
MMIO page
unmap virtual page for tID

Figure 4.3: Flow Chart for the Simulation Driver

When a virtual machine instance is attached to the simulation interface with ioctl calls, the simulation driver will request a MMIO page address with the virtual machine thread id (tID) from the virtualized device driver, through the simulation interface, which will respond with a MMIO page address from the available MMIO address space, using a different page for any different virtual machine instance attached to it. Since the user-level virtual machine has no privilege to directly access physical MMIO page, the simulation driver needs to map the physical MMIO page to a process virtual page and return the virtual page address mmio to the virtual machine, which will be used for the virtualized simulation counter. That prevents different virtual machine instances from interfering with each other because a virtual machine can only access the MMIO page mapped to its process virtual page. The detachment from the simulation interface will lead to release of the MMIO page for reuse.

Our scheme leverages the Operating System (OS) to reserve a physical memory address to the scc, through the simulation driver, which is then mapped to the user virtual address space. The scc has to reside in physical memory; otherwise, if defined in virtual memory address space, it could occupy different physical memory positions during execution, due to swapping. In this scenario, another mechanism would have to be implemented to inform the simulator that the virtualized counter is residing in another virtual address, which is not a trivial task. In our design, we implement a java native function init that is called to allocate the address for the scc, as shown in Figure 4.3 on page 75.

### 4.3.3 Virtual Machine Modification

We modify the virtual machine implementation for accelerating managed code execution as shown in Listing 4.2. The virtual machine first tries to open the OS device created by the simulation control driver with an OS call open. If the OS device cannot be opened due to either there being no simulation driver or because the driver is not correctly installed, an error message will be shown, and all the calls to the simulation control API will become

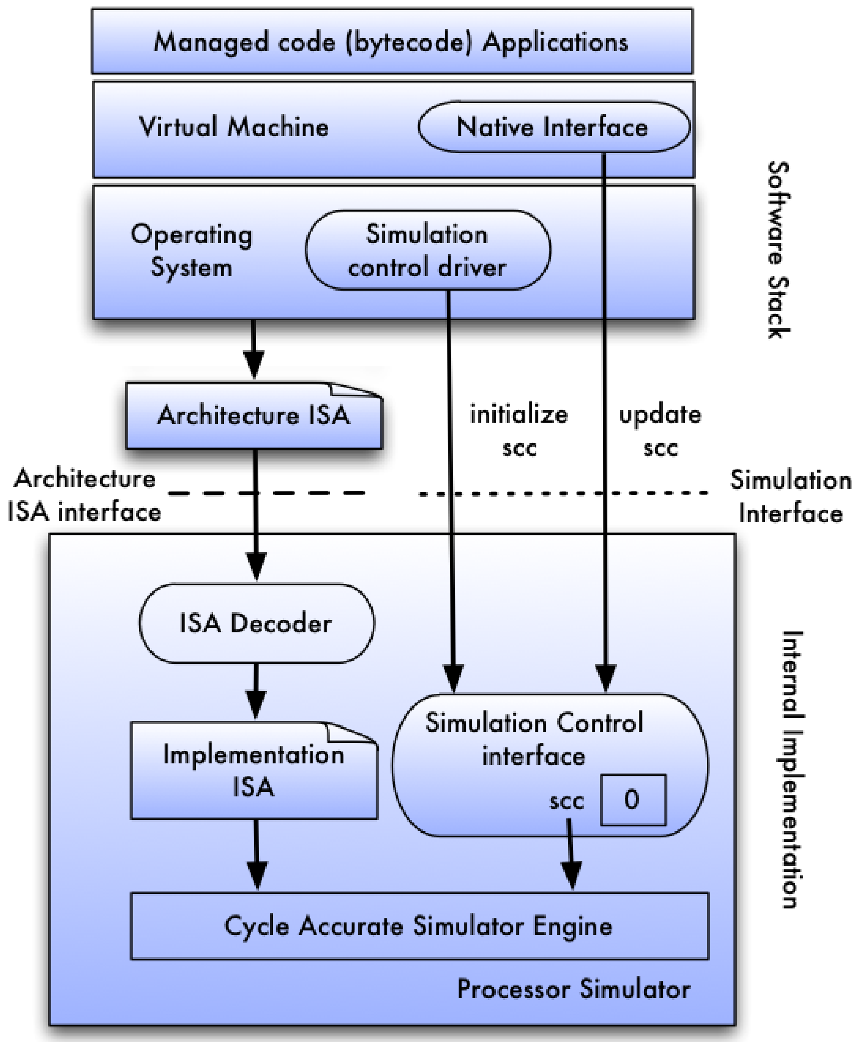


Figure 4.4: Simulation Control Interface

no-ops. If the OS device is opened successfully, the simulation interface will be initialized and ready for use by the bytecode.

The novelty of our technique is that it does not require: 1) modifications to the syntax of the language the application is written in; 2) modifications to the static compiler, since there is no change in language syntax; or 3) modifications in the compiler to generate special instructions into the output binary. For dynamic languages that execute in a VM (ex. Java, Javascript, Python, Ruby, etc.), there is no need to change the portable bytecode (Intermediate) representation, the interpreter, the JIT compiler, or the VM runtime. Our

```
VM_run() {
   execute bytecode in
   the virtual machine;
}

VM_init () {
   // open the simulation device
   if ( (fd = open ('/dev/simDevice')) == -1) {
       printf("Could not open Simulation Device \n");
       VM_run();
   }
   else {
       initialize Simulation Control Device
       // attach to the simulation interface and
       // return a memory-mapped I/O address

       mmio = ioctl ( fd, ATTACH, ACB);
       // run the application bytecode
       VM_run();

       ioctl (fd, DETACH, mmio);
       close (fd);
     }
}
```

Listing 4.2: Virtual Machine Initialization

```
// Virtual Machine global variable
long *scc;

// Native call to initialize scc
JNIEXPORT void JNICALL Java_SCC_init
  (JNIEnv *env, jobject obj, jlong value) {

  // GetSCCAddress returns the virtual address for scc
  scc = ((*env)->GetSCCAddress(env));
  (*scc) = value;
}

// Native call to increment scc
JNIEXPORT void JNICALL Java_SCC_increment
  (JNIEnv *env, jobject obj) {
  (*scc)++;
}

JNIEXPORT long JNICALL Java_SCC_get
  (JNIEnv *env, jobject obj) {
  return (*scc);
}
```

Listing 4.3: JNI implementation for Simulation API

simulation control technique also enables fast creation of cross-architecture simulation points at a certain marker value (marker boundary).

### 4.3.4 Modifications to the Architecture Simulator

We now show how to modify the architecture simulator so that it may monitor the byte-code execution according to the updates to the *scc*. Our novel technique simplifies the modifications required in the architecture simulator, since the simulator only needs to monitor the updates to the *scc* value, using it to control the application execution during the energy-performance analysis and processor validation. Our scheme is of special interest in the design exploration of co-designed processors [Kre03], as they have additional layers of interpretation and translation in their internal runtime.

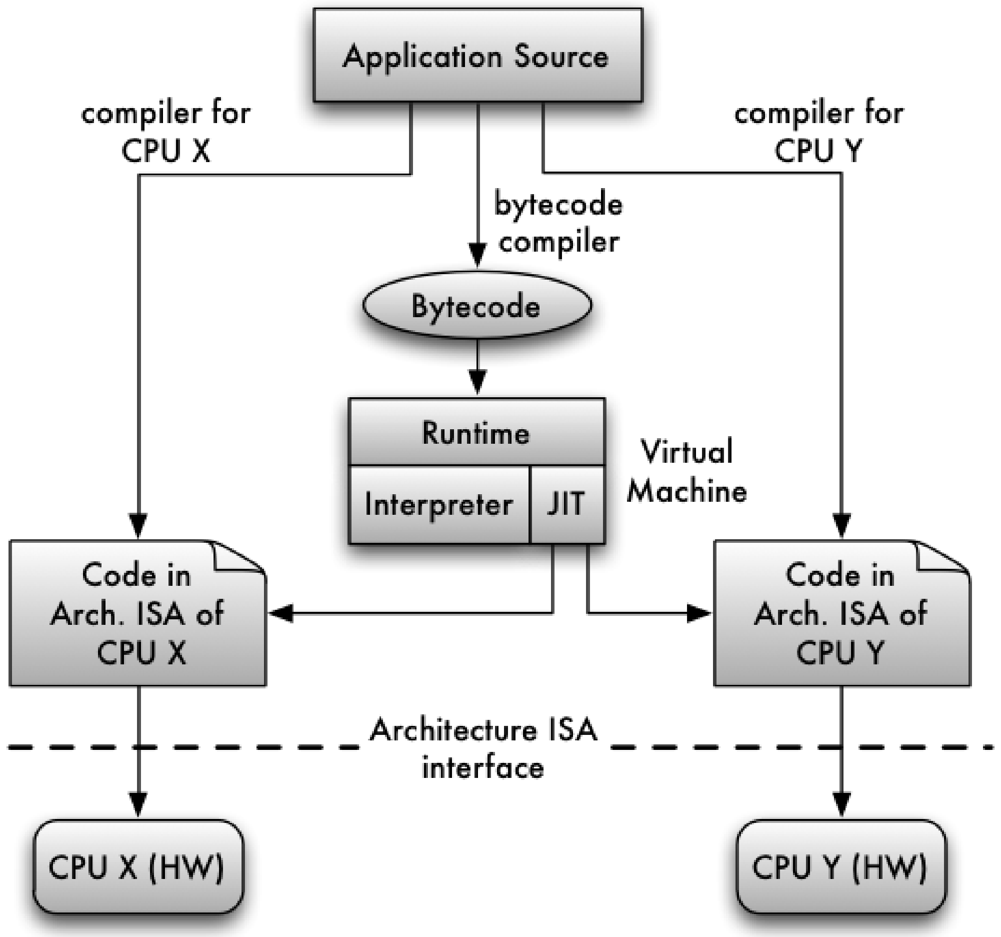


Figure 4.5: Application source compilation into binary and bytecode format

### 4.3.5 Application Instrumentation

In Figure 4.6 on page 79 we show how our novel marker implementation can be used to control the simulation execution. The marker is defined and incremented at application source level. No modification is required in the compiler, VMs or in the Architecture ISA interface, due to the way we implement the marker, which is guided by the following principles:

1. It does not require extensions to the syntax of the source language, nor the handling of special intrinsic functions.

2. There is no special instruction being generated in the intermediate representation (byte-code) handled by the Virtual Machine, nor does it require special runtime support.

3. The Architecture ISA is not modified, so there is no need to modify the underlining hardware/Architecture simulator to support the marker.

In order to implement the virtualized control technique, the application source has to be modified to instantiate an object of the SCC class, which is defined by the accelerator API and encapsulates the creation and use of a virtualized marker in physical memory by providing functions to initialize (constructor) and increment() the marker (*scc*) value.

In the implementation of the API class SCC, a special accelerator device, which is initialized by the OS, provides the unique physical memory address where the *scc* resides. The *scc* cannot be allocated in Virtual Memory address space, since the page swapping can change the physical memory address where it resides during execution. This happens when the paging supervisor frees the page in primary storage that holds the marker and swaps it out to secondary storage. Once the marker is accessed again, a page fault will be raised, and the page is swapped back into a different memory page. This results in a different physical address for the *scc*, which makes it impossible for the simulator to track its value.

The SCC class constructor will get the *scc* address (non-Virtual) by calling the accelerator device. Once the address is returned, the constructor sets the initial marker value to 0 (zero). Once the *scc* is defined, the application developer has to do add calls to the increment() method in places of interest. The calls to the increment() method are usually placed at the beginning and at the end of a region of interest for performance evaluation.

Now that the application is modified, it only needs to be recompiled, using the existing compiler for the source language. As aforementioned in this chapter, no modifications are needed in the underline hardware/software stack (compiler, VM, nor the Architecture ISA).

In order to control the application execution, the simulator has to be extended to monitor the increments of the *scc*. Since the accelerator device is also visible to the simulator, it can also provide the simulator with the same physical address where the *scc* resides, so that the simulator can use it to control the workload execution based on *scc* values.

The only modification to the simulation environment is the addition of commands to control the execution based on the marker value. For example, in Figure 4.6 on the next page, the command "run-to-scc == 1", would start the execution and stop immediately when the *scc* is incremented to 1. Whenever the *scc* reaches the desired value, the simulation execution is stopped, and existing commands can be executed to dump specific hardware counters into a log file, which later can be processed by an offline tool to roll-up the performance-per-watt data.

Note that the same simulator program could be used to simulate two different co-designed processors, so that the modifications in the simulation control unit need be implemented only once.

Although not shown in Figure 4.6 on the following page, our technique can also facilitate the development and the energy-performance analysis of simulators for processors that implement the "Code in Architecture ISA Z". In this case, the marker would also control the simulation execution, and no modification to the Architecture ISA would be needed.

## 4.4 Evaluation

### 4.4.1 Experiment Setup

We evaluated our bytecode simulation scheme using the Android Dalvik Virtual Machine in a research x86 processor design running on a product-quality timing-accurate simulator.

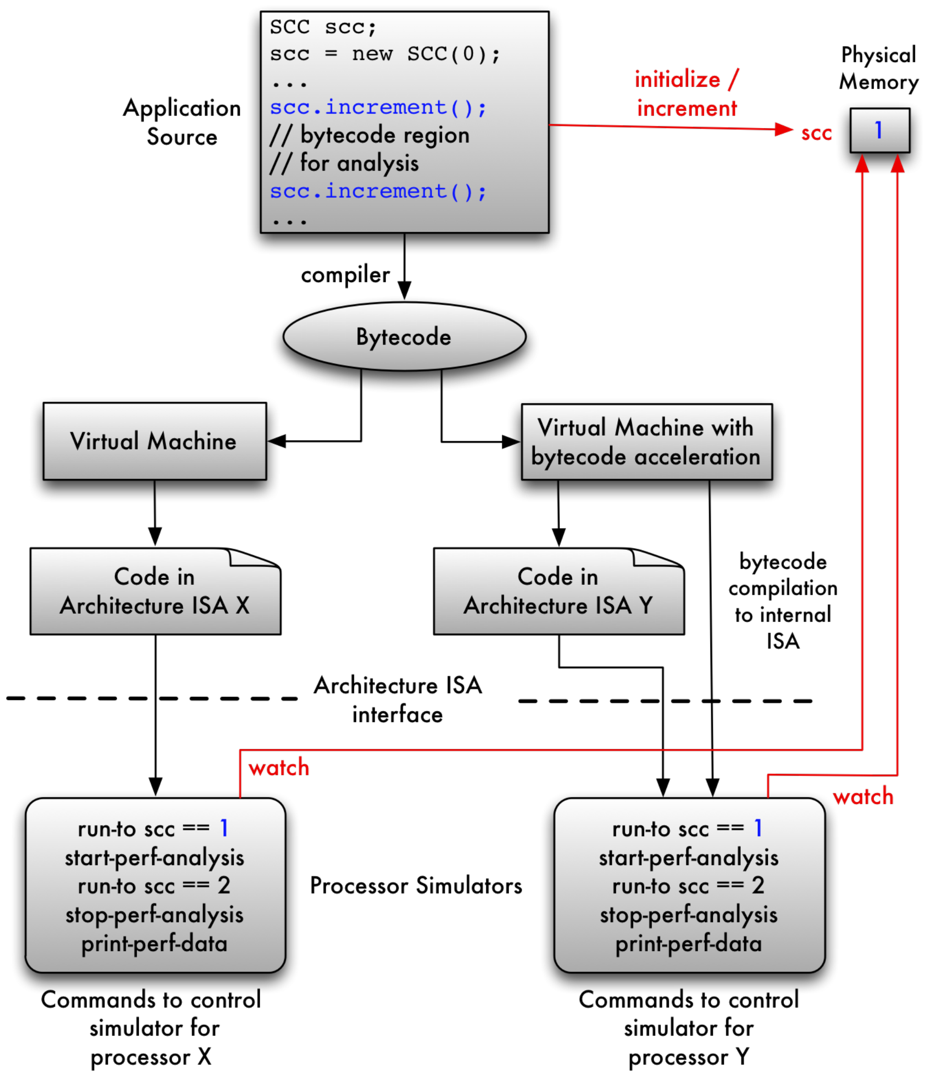


Figure 4.6: Application Instrumentation and Simulation Control

We experimented with a x86 version of the Android system for Medfield phones. We implemented a simulation driver and plugged it into the Android OS. We modified the Dalvik virtual machine to open the simulation driver and initialize the native call interface. The JNI call interface of the Dalvik VM was extended to include one call to increment the *scc* value. Overall, we added 150 lines of code for the simulation driver and changed 50 lines of code in the Dalvik virtual machine in order to initialize the simulation interface as well as to implement the JNI call to control the SCC.

### 4.4.2 Workloads

We used the industry standard CaffeineMark Android benchmark for evaluating the overhead of our simulation control technique. The CaffeineMark is a series of synthetic benchmarks that measure the performance of Java programs running in various hardware and software configurations [Cor97]. It integrates a series of benchmarks for the Android system into a comprehensive benchmark suite.

To compare Dalvik bytecode execution on two processor simulators, we used our scheme to put virtualized bytecode markers (calls to increment the *scc*) into CaffeineMark applications in order to measure the performance and energy consumption for the same bytecode snippets. For each benchmark program, we collected data on the execution of 25 randomly selected snippets and averaged over them so that we could report the data for that program. For each measurement, we ran at least 1 billion instructions to warm up the just-in-time compilation in the functional execution of the simulator, then at least 100 million instructions to warm up the simulator microarchitecture states, and at the end collected performance and energy data on the execution of at least 300 million instructions between markers. The marker was used on both systems to delimit the boundaries of JIT compilation, micro-architecture, and the actual energy/performance evaluation.

Figure 4.7 on the next page shows the execution overhead for CaffeineMark. Overall, the additional calls to increment the marker incurs less than 3% execution overhead.

Our technique can also be used to measure the amount of dynamic code (bytecode) versus the native code executed, which is native code executed in native libraries and native code called via JNI calls. In this scenario, the bytecode marker is be used to indicate when the execution engine is running bytecode versus native code. The simulator software is extended to increment the counter for bytecode execution, or the native code execution, according to the bytecode marker value. With our scheme, this measurement was straight-

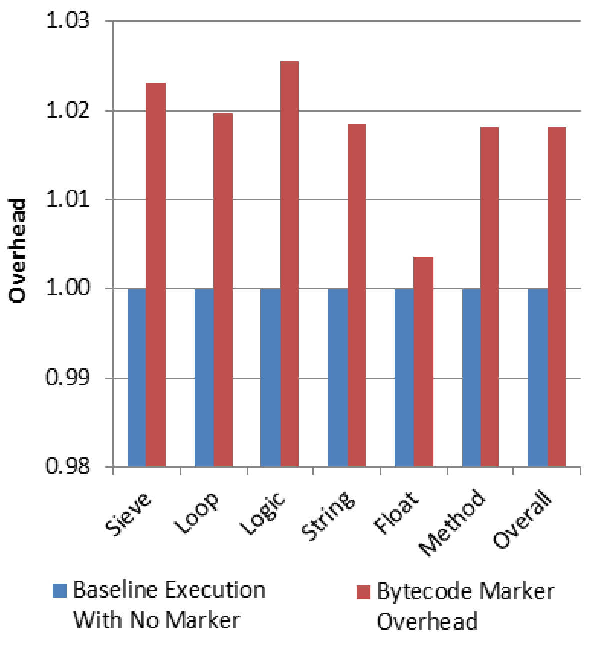


Figure 4.7: Execution overhead for CaffeineMark with bytecode marker

forward and allowed us to get a better understanding of the benchmark we used to collect the performance described in the next chapter.

| Benchmark | Dynamic Code | Native Code | Total | % Dynamic |
|---|---|---|---|---|
| **0xBench Linpack** | 6.00E+08 | 4.00E+08 | 1.00E+09 | 60.00% |
| **0xBench Scimark** | 1.36E+10 | 1.34E+09 | 1.49E+10 | 91.01% |
| **EEMBC AndEBench** | 5.49E+09 | 2.64E+09 | 8.13E+09 | 67.57% |

Table 4.1: Percentage of Dynamic Code Execution on Android Benchmarks

We also used our marker implementation to measure the ratio of bytecode execution for CaffeineMark. Figure 4.8 on the following page shows the ratio of dynamic code (bytecode) executed in CaffeineMark. Overall, 95% of the instructions in CaffeineMark run in bytecode. The majority (over 90%) of code executed in the String benchmark is native code, not dynamic code (bytecode), since one of the objectives of this benchmark is to test the performance of garbage collection. This finding indicates that the String benchmark is not suited to evaluate a bytecode acceleration framework such as AccelDroid, since most of the execution time is spent on native code.

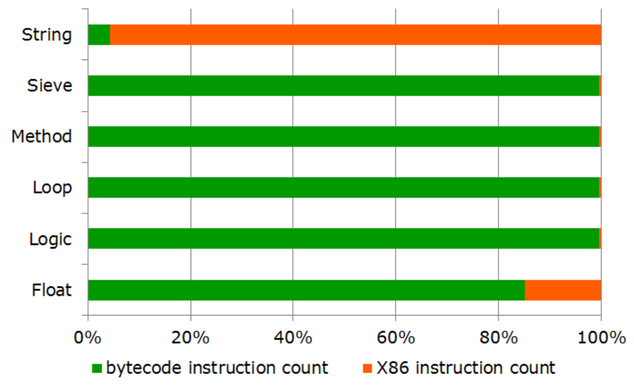


Figure 4.8: Bytecode vs. Native Execution for CaffeineMark

## 4.5 Related Work

In an architecture design exploration, it is necessary to take one instance of a program with a given input, and simulate its performance over many different configurations for any given architecture feature. Simulating the full execution of a bytecode application benchmark on a cycle accurate simulator can take days or even weeks to complete.

SimPoint [SPC01, PHC03b, SPHC02] automates the process of picking simulation points using an offline phase classification algorithm based on k-means clustering, which significantly reduces the amount of simulation time required.

It is known that whole program simulation does not represent the majority of the program's behavior because the code that is executing is often cold code allocating and clearing the data structures to be used. To skip over the initialization phase, the simulator usually fast-forwards the application to a given point in execution, and then starts the simulation from there [PLP+07]. Ideally the code that is executed should be a representative of the whole execution.

An alternative to fast-forwarding is to use checkpointing to start the simulation of a program at a specific point. With checkpointing, the program is executed up to a given point and the state is checkpointed (saved) so that the simulation later can be re-started from this point. One advantage of checkpointing is the ability to simulate all of the samples in parallel, thus obtaining very fast results using distributed computing [Lau94]. One drawback of this technique however is that the checkpoints can become quite large since the contents of main memory need to be saved along with the architectural state.

Binder at al. [BH04] applied bytecode instrumentation in order to monitor and control resource consumption in standard Java Virtual Machines (JVMs), and to generate calling context-sensitive profiles for performance analysis. In [BHM07] extends the framework to allow dynamic instrumentation of bytecode. Our approach is similar since the application has to be modified to insert API calls to monitor the counter.

The Pin [LCM+05] dynamic instrumentation system is a easy-to-use, efficient, and transparent system to instrument embedded applications to perform tasks such as profiling, performance evaluation, and bug detection, without the need for application source code. Pin uses dynamic compilation of instrument executables and cannot be used to instrument the platform independent bytecode format. Furthermore, Pin significantly slows the application down by an average of 2.5 times [LCM+05].

## 4.6 Conclusion and Future Work

At the heart of computer architecture design exploration is the need to precisely evaluate the impact of novel architectural features on the overall energy/performance efficiency of a system. Modern mobile applications execute on a virtual machine that can run on top of different hardware platforms. For efficiency, the bytecode is dynamically compiled to the native ISA of each hardware platform. To help compiler and architecture researchers in simulating managed code applications on new processor design exploration, this chapter proposed a new mechanism to facilitate the control the execution and energy/performance analysis of bytecode applications on processor simulators. Our technique requires the workload applications to be instrumented, using our proposed API, in order to monitor a special marker at points of interest for performance evaluation. The simulation control marker is allocated in a region of physical memory address space that is visible to both the Virtual Machine and the simulation software. An OS simulation driver is used to define the control marker in physical memory. Our scheme requires significantly less implementation effort to modify the virtual machine, the OS, and the architecture simulator software.

We implemented the scheme proposed in this chapter in a state-of-art architecture simulator for an Out-Of-Order processor. This enabled us to quickly perform the energy/performance analysis of our bytecode acceleration framework implemented on top of a OOO processor, which is the subject of the next Chapter. This is strong evidence of the value of our scheme for architecture design exploration, since it allows the same bytecode application to be evaluated in different architecture designs, possibly having completely different ISAs, reducing the time-to-market, which is an important aspect in mobile processor designs.

Our simulation control scheme can be extended to monitor the performance-per-watt of a released processor. To achieve this, the monitoring tool would watch the marker values, collecting hardware counters when the control marker reaches a specif value during execution. In this approach, a virtual device driver has to emulate the OS simulation driver, so that the OS and Virtual Machine can interact via the simulation control interface, as described in Figure 4.3 on page 71.

# Chapter 5

# Bytecode Acceleration in a Out-of-Order Co-Designed Processor

*"Perfecting oneself is as much unlearning as it is learning."*

Edsger Dijkstra

In Chapter 3 on page 29, we presented AccelDroid, a bytecode acceleration scheme that speeds up bytecode execution by leveraging the hardware features exposed by the internal ISA of a co-designed, in-order processor, the Transmeta Efficeon. In this chapter we explore the benefits of direct compilation of bytecodes into the internal ISA of a modern Out-of-Order (OOO) processor, which employs an efficient floating point unit compared to Efficeon. We use a state-of-art, cycle accurate processor simulator to perform our energy and performance evaluations.

## 5.1 Motivation

In Chapter  3 we presented AccelDroid, one implementation of our bytecode acceleration interface designed for the Dalvik Virtual Machine (DVM), which allows us to run bytecode through the internal implementation ISA (I-ISA) of a co-designed processor. AccelDroid is designed in support of the thesis statement of this dissertation, which aims at exploring the effects, on energy and performance efficiency, of JIT optimizations that are made possible when the ISA is exposed to the JIT compiler through the acceleration interface.

We implemented and evaluated AccelDroid on top of an in-order processor, the Transmeta Efficeon. An in-order processor processes the instructions in the order that they appear in the binary (according to the sequential semantics of the instructions), relying on the compiler to provide the best schedule for the instructions, whereas an Out-Of-Order(OOO) processor processes the instructions in an order that can be different (and usually is) from the one in the binary. The purpose of executing instructions out of order is to increase the amount of instruction level parallelism (ILP) by providing more freedom to the hardware to dynamically choose which instructions to process in each cycle. The OOO engine executes the instructions in an order that is governed by the availability of input data, rather than by their order in the executable's binary. Obviously, OOO processors require more complex hardware than in-order ones, since the scheduling of the instructions is performed dynamically, in hardware, as the program executes, which results in increased energy consumption.

A dynamic instruction issue scheme is a common trend in today's high performance mobile processor design due to its potential to exploit Instruction-Level Parallelism (ILP) at runtime, for applications whose behavior is hard to predict at compile time. However, the hardware support required by a dynamic instruction issue scheme is one of the most critical components of modern microprocessors, both from the delay and energy dissipation standpoints, which may affect the cycle time [PJS97], and the amount of the energy consumed by the processor [CG01]. Processor designers face the challenge of delivering high single-thread performance while at the same time not increasing the complexity of the design.

It is generally challenging to design an acceleration interface for a hardware accelerator due to the complicated synchronization and communication between CPU and the accelerator, and to precisely handle the exceptions. Our acceleration interface is designed to run bytecode through uop runtime on the CPU pipeline, not on a hardware accelerator. So we do not need to consider any synchronization or communication issues in our acceleration interface design. In this chapter we implement AccelDroid on a research OOO x86 processor and show that the programming for acceleration only requires 150 lines of code for an OS acceleration driver and 30 lines of code change in the existing Dalvik VM implementation. Most of the implementation is in the internal uop runtime, which is invisible to programmers. To summarize, this chapter makes the following main contributions:

- We present a novel acceleration scheme for bytecode, which allows flexible and efficient acceleration of the general-purpose bytecode execution in the state-of-art CPU design.
- We identify many interesting challenges in our acceleration scheme and provide novel SW/HW techniques for handling them.
- We implement our overall acceleration scheme for Android Dalvik bytecode on a research x86 processor and evaluate the benefits with solid performance and energy results.

## 5.2 Acceleration Interface Design

In this section, we discuss our acceleration interface design with the following goals:

*ISA-Compatibility*: The acceleration interface does not require any new architecture ISA support.

*Virtual Machine Simplicity and Flexibility*: The acceleration interface only needs minor changes to the existing VM implementations for accelerating the bytecode execution, and is flexible enough to support different VM implementations.

*OS-Transparency*: The acceleration interface does not need any change to the existing OS for accelerating bytecode execution, except a simple kernel mode acceleration driver that is plugged into the existing OS.

We next discuss the detailed acceleration interface design aimed at achieving the above goals. Since our uop runtime runs bytecode with an acceleration interface, we simply call the bytecode execution through uop runtime of the accelerated managed code execution.

### 5.2.1 ISA-Compatibility

For ISA-compatibility, our acceleration interface is supported by an OS acceleration driver via privileged architecture I/O instructions (e.g., x86 in/out instructions). The virtual machine accelerates bytecode execution through OS calls to the acceleration driver. Frequent switches between the bytecode execution and architecture ISA code execution, in interactive web and mobile applications, can result in large execution overhead due to the frequent and expensive OS calls. For efficiency, our acceleration interface can also be accessed via an architecture memory-mapped I/O (MMIO) instruction. The OS calls are only used for initializing the MMIO access, which happens at VM start-up. After that, the VM can trigger accelerated bytecode execution via non-privileged MMIO access without using the expensive OS calls.

### 5.2.2 Virtual Machine Simplicity and Flexibility

It is challenging to design an acceleration interface for a hardware accelerator due to the complicated synchronization and communication between CPU and the accelerator. Our acceleration interface is designed to run bytecode through uop runtime on the CPU pipeline, not on a separate hardware accelerator unit. So we do not need to consider the synchronization and communication issues in our acceleration interface design. The accelerated bytecode execution is simply triggered by an architecture instruction accessing MMIO, which switches the execution from architecture ISA to accelerated bytecode execution. The architecture ISA execution mode is resumed once it returns from the accelerator, as shown in Figure 5.1. While executing in acceleration mode, the code can directly access and update the CPU architecture states without the need for explicit data communication between the architecture ISA code execution and the accelerated bytecode execution.

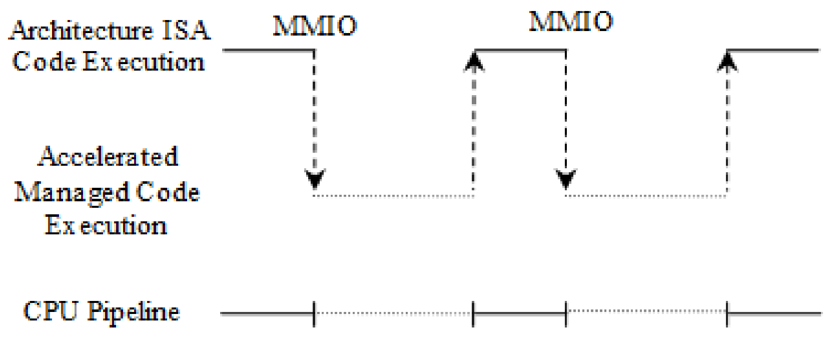

Architecture ISA Code Execution
MMIO
MMIO
Accelerated Managed Code Execution
CPU Pipeline

Figure 5.1: Accelerated Bytecode Execution

Since the execution in acceleration mode directly accesses and updates the CPU architecture states, including general-purpose registers, process virtual memory, etc., in order to flexibly support different virtual machine implementations we have defined an Acceleration Control Block (ACB) data structure. ACB specifies the mappings from all of the bytecode states to the CPU architecture states such that the accelerated bytecode execution will access and update the corresponding CPU architecture states. For example, if the ACB maps the bytecode register r0 to the x86 register EAX, then the managed code that updates r0 will update EAX in the accelerated bytecode execution. The precise CPU architecture states controlled by ACB are maintained only at the moment of switching between architecture ISA code execution and accelerated bytecode execution. So the uop runtime can still freely run the bytecode with its internal microarchitecture states independent of ACB by just copying-in/copying-out the precise CPU architecture states according to ACB during the switching between architecture ISA code execution and accelerated bytecode execution.

### 5.2.3 OS Transparency

Hardware interrupts during the accelerated bytecode execution may need OS handling. For OS-transparency, we designed the bytecode accelerator to switch to architecture ISA code execution on hardware interrupts before triggering the OS interrupt handling. Then the interrupts can be handled as normal during the architecture ISA code execution without OS change. Hardware exceptions triggered by the accelerated bytecode execution also cause the switching to architecture ISA code execution. However, after switching to the architecture ISA code execution, the exceptions are masked from OS exception handling as the OS has no knowledge of the exceptions triggered by the accelerated bytecode execution. Instead, the following architecture ISA code execution will rerun the bytecode instruction that triggers the exception through the regular managed runtime without acceleration. Then the exception can be triggered and handled during the regular virtual machine execution without OS change.

## 5.3 Programming for Acceleration

In this section, we discuss the system programming for acceleration, i.e., change of existing virtual machines for accelerating bytecode execution with an OS acceleration driver. We will discuss the internal implementation for acceleration in Section 5.4 on page 93.

### 5.3.1 Virtual Machine

We modify the virtual machine implementation for accelerating bytecode execution as shown in Figure 5.1 on the next page. Existing VMs run the bytecode through interpretation or just-in-time compilation. We implement an acceleration virtual machine, acceleration_VM, for accelerating bytecode execution. The acceleration virtual machine first attempts to open the OS device supported by the acceleration driver (discussed in Section 3.2.1 on page 32) installed, by using an OS call open. If the OS device cannot be opened due to either there being no acceleration support in the CPU or because the OS acceleration driver is not correctly installed, the VM proceeds to the regular virtual machine execution. If however the OS device is opened successfully, it switches the execution to the bytecode acceleration mode.

The acceleration virtual machine makes an OS call ioctl to attach the virtual machine instance to the acceleration interface, passing the ACB (see discussion in Section 5.2.2 on page 89), and returning a memory-mapped I/O (MMIO) address mmio. After that, the acceleration virtual machine triggers the accelerated bytecode execution with a MMIO access to address mmio. Multiple acceleration virtual machine instances may be attached to the acceleration interface at the same time, which may use different ACBs for controlling the accelerated bytecode execution. Different acceleration virtual machine instances will get different MMIO addresses when being attached to the acceleration interface for

```
VM () {
        run bytecode with interpretation or just-in-time
                compilation in a regular virtual machine
}

acceleration_VM () {
    // open the OS acceleration device
    if ( (fd = open ('/dev/acceleration')) == -1) {//open fail
        VM();
    }
    else {  // run bytecode with acceleration
        initialize ACB

        // attach to the acceleration interface and
        // return a memory-mapped I/O address
        mmio = ioctl ( fd, ATTACH, ACB);

        // accelerated bytecode execution
        while ( not end of program ) {
             trigger accelerated bytecode execution with
                 a MMIO access to address mmio
             // upon return from accelerated
             // bytecode execution
             if ( not end of program )
                 run one bytecode instruction with
                   interpretation in a regular virtual machine
        }

        ioctl (fd, DETACH, mmio);
        close (fd);
     }
}
```

Listing 5.1: Existing DVM Implementation

distinguishing them by the underlying implementation for acceleration (see Section 5.4).

The accelerated bytecode execution may be stopped due to HW interrupts/exceptions (see Section 5.2.3 on page 90). After returning from the accelerated bytecode execution, the acceleration VM runs one bytecode instruction using the interpreter running on top of the OS, executing architectural ISA code. In this way, the bytecode instruction that triggers a HW exception (see discussion in Section 5.2.3 on page 90) will be handled during the interpretation in non-acceleration mode, just like the bytecode execution in a regular virtual machine without bytecode acceleration. After the interpretation of one bytecode instruction, the accelerated bytecode execution is triggered again with the MMIO access to address mmio, and this process repeats until the end of the program. The interpretation in regular virtual machine also allows flexible uop runtime implementation to support only a subset of frequently-executed bytecode instructions. The corner-case non-supported bytecode instructions will run through the interpretation in non-acceleration mode, which simplifies the implementation of the accelerator in the uop runtime.

## 5.4 Internal Implementation for Acceleration

In this section, we will discuss the internal implementation for our acceleration, i.e., the uop runtime to run bytecode according to the acceleration interface, and the corresponding hardware supporting it. We will discuss these in detail in the following subsections.

### 5.4.1 Hardware Support for Acceleration Interface

Similar to the HW/SW co-designed processor [Kre03, EA97], uop runtime reserves internal memory invisible to OS/application to store code and data. To support the architecture

I/O instructions in the acceleration interface, we implemented an internal microarchitecture trap to trigger the uop runtime execution on all privileged architecture I/O instructions.

It would be expensive to implement an internal microarchitecture trap on all instructions accessing MMIO. We implemented a special microarchitecture register for holding the range of MMIO address space reserved for the acceleration interface such that only the instructions accessing the reserved MMIO address space will trigger an internal microarchitecture trap. The uop runtime will set up the register with the reserved MMIO address space passed from the acceleration driver (see discussion in Section 3.2.1 on page 32). Figure 5.2 on the following page shows the trap handling in uop runtime. Unlike the trap handling in OS, the internal microarchitecture trap handling in uop runtime can be implemented in a very lightweight manner by using reserved internal microarchitecture registers without the overhead of register context switch.

The bytecode execution through uop runtime needs to stop in cases of HW interrupts and exceptions. So we also implemented an internal microarchitecture trap on all HW interrupts/exceptions during the uop runtime execution. The trap handler in our uop runtime will handle all the HW interrupts / exceptions by stopping the bytecode execution through uop runtime and switching to the architecture ISA code execution, as shown in Figure 5.2 on the next page.

## 5.5 Uop Runtime

Similar to the regular virtual machine, our uop runtime just-in-time compiles the bytecode regions into the microarchitecture ISA code regions (we call them translations) and store them in an internal microarchitecture translation cache in order to avoid recompila-

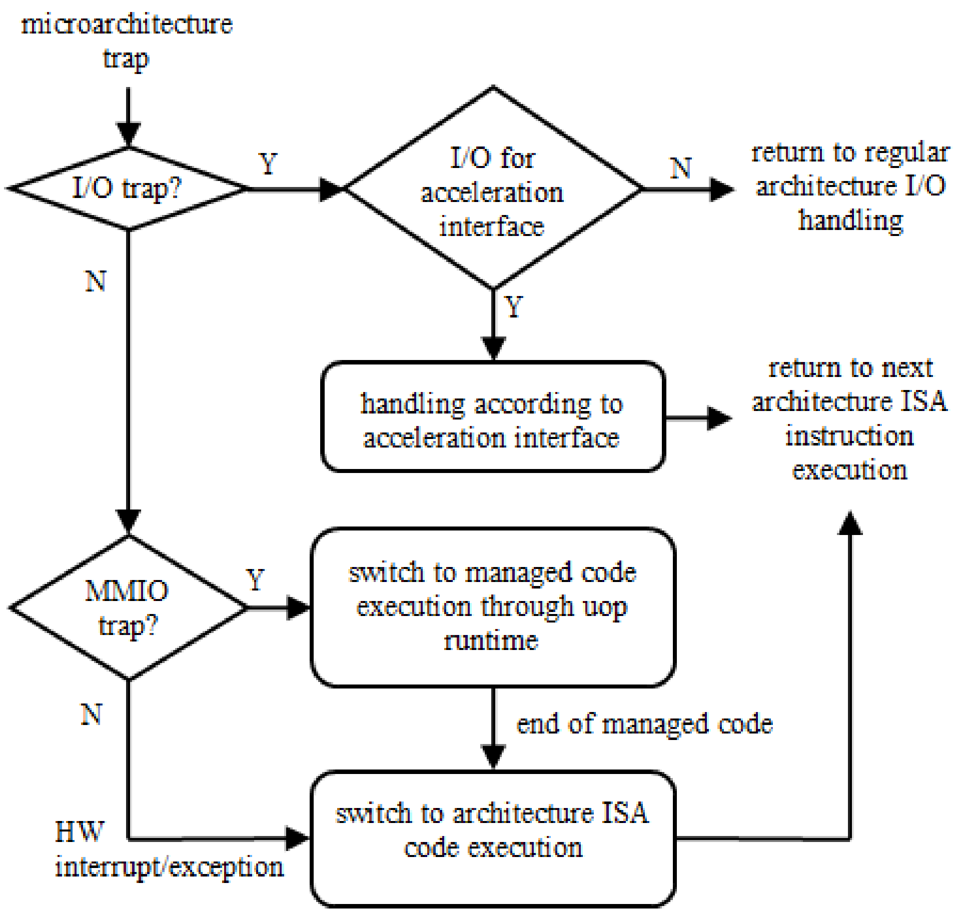

microarchitecture trap
I/O trap?
Y
N
I/O for acceleration interface
N
return to regular architecture I/O handling
Y
handling according to acceleration interface
return to next architecture ISA instruction execution
MMIO trap?
Y
switch to managed code execution through uop runtime
N
end of managed code
HW interrupt/exception
switch to architecture ISA code execution

Figure 5.2: Flow Chart for Trap Handling in Uop Runtime

tion when the same bytecode region is executed again. We also implemented a bytecode interpreter in the uop runtime such that the initial execution of bytecode goes through interpretation to collect runtime profiling information. Only the hot bytecode regions are compiled to the I-ISA for efficient execution.

However, there are several implementation issues in the uop runtime, which do not exist in the regular virtual machine. We discuss these issues in the following subsections.

### 5.5.1 Handle Multiple Virtual Machine Instances

Multiple virtual machine instances may be attached to the acceleration interface at the same time. So, unlike the just-in-time compilation in a regular virtual machine, which only handles the translations for the bytecode regions in current virtual machine instances, our uop runtime needs to distinguish translations belonging to different virtual machine instances. Since a virtual machine instance triggers bytecode execution through uop runtime with a unique MMIO address (see discussion in section 3.1), the uop runtime distinguishes the translations belonging to different virtual machine instances by using the MMIO addresses that trigger their corresponding bytecode execution.

### 5.5.2 Handle HW Interrupt and Exception

For the internal microarchitecture trap triggered by HW interrupts and exceptions, the uop runtime needs to restore the precise architecture states according to ACB at a bytecode instruction boundary before switching to architecture ISA code execution. This brings restrictions to the optimizations in our just-in-time compilation for bytecode. Previous works [DGB+03, NRS+07, PTBC00, NDZ10] show that the hardware atomicity support can greatly improve the optimization by putting the optimized code into an atomic region for speculative execution. In case of HW interrupts / exceptions, the atomic region execution can be rolled back to the precise states at the entry of the region. For exceptions, the precise states need to be immediately before the bytecode instruction that triggers the exception. So, after rolling back, our uop runtime reruns the bytecode in the region one instruction at a time until the instruction that triggered the exception is executed.

### 5.5.3 Handle Non-Supported Instruction

The regular virtual machine relies on underlying OS support for bytecode execution. Our uop runtime runs below the OS and cannot handle system operations such as OS I/O, OS thread management, OS memory management, etc. There are two ways to run system operations in the bytecode applications. First, the bytecode may support a special native library call instruction (e.g., Java Native Interface call or JNI call) to run system libraries (e.g., I/O library). Second, a special bytecode instruction may be used for a particular system operation such as the synchronization instructions monitor-enter/monitor-exit in Dalvik bytecode, whose execution may yield for OS thread context switch. In both cases, our bytecode execution (after they are translated into microarchitecture ISA code) continues through uop runtime stops and switches to architecture ISA code execution. Then the interpretation in the regular virtual machine (see Figure 5.1 on page 92) will run the JNI call or the special bytecode instructions leveraging the OS support.

Certain bytecode execution may highly depend on a particular virtual machine implementation which is not flexible enough to run by our uop runtime. For example, different virtual machines may implement different garbage collection algorithms. To flexibly support virtual machines implementing different garbage collection algorithms, our bytecode execution through uop runtime also stops and switches to architecture ISA code execution on the bytecode instructions that may trigger garbage collection (e.g., memory allocation/free instructions).

### 5.5.4 Handle Floating Point Operations

The OOO processor used in our implementation contains the Fused Multiply-Add (FMA) floating point instruction. FMA increases the performance by reducing the latency of dependent add-multiply operations, increasing the precision by computing the result as an indivisible operation with no intermediate rounding.

We modified the just-in-time compilation in uop runtime to generate efficient microarchitecture ISA code for bytecode traces with floating-point operations, leveraging the hardware support available in the I-ISA of the underlying processor, such as the FMA instruction. The just-in-time compilation in the regular managed runtime does not have the knowledge necessary to generate efficient code at the x86 level. Also, since we leverage the internal HW speculation feature [NRS+07, PTBC00], converting the rarely used conditional branches into assertions, we create larger basic blocks which allow for more speculative optimizations, such as FMA, to be performed across these asserts. In case an assertion fails, the atomic region is rolled back, and the execution switches back to the architectural ISA mode.

## 5.6 Experiment Setup

We evaluated our managed code acceleration scheme for Android Dalvik bytecode in a research x86 processor running on a product-quality timing-accurate simulator. The baseline design of the processor implements an out-of-order pipeline competitive to x86 mobile processors in terms of energy, performance and die area, with an average IPC around 1.5 for SPEC2006. Table 5.1 on the next page lists the important microarchitecture parameters in our research x86 processor

We experimented with an x86 version of an Android system designed for Medfield phones. The baseline Dalvik virtual machine in the Android system implements a managed runtime with just-in-time compilation for x86 processors with a CaffeineMark score [Cor97]

|  |  |
|---|---|
| **L1 I-Cache** | 32 KB |
| **L1 D-Cache** | 24 KB |
| **L2 Cache** | 256 KB |
| **Reorder Buffer** | 128 entries |
| **Reservation Station** | 64 entries |
| **Branch Ordering Buffer** | 24 entries |
| **Load Buffer** | 40 entries |
| **Store Buffer** | 24 entries |
| **Physical Integer Register** | 128 |

Table 5.1: Important Microarchitecture Parameters

of around 10500 on Medfield phones. We implemented an acceleration driver and plugged it into the Android OS. We modified the Dalvik virtual machine to accelerate Dalvik byte-code execution with the acceleration driver. Overall, we added  150 lines of code for the acceleration driver and changed  30 lines of code in the Dalvik virtual machine. We ported the just-in-time compilation in the Dalvik virtual machine to our uop runtime, with code generation targeting the microarchitecture ISA. We implemented the whole tool-chain for microarchitecture ISA to compile our uop runtime source code to the microarchitecture ISA code, so the simulation measures all of the code executions, including the just-in-time compilation in the uop runtime.

| **HW Features** | **Benefits** |
|---|---|
| Atomic Region | Rollback in Speculative Optimization |
| Assert | Control Speculation |
| HW Alias Detection | Data Speculation |
| Fast Internal trap | Reduce Mis-Speculation Overhead |
| Large Microarchitecture Register File | Efficient Register Allocation |
| Control Transfer Support | Fast Code Control Transfer |
| Profiling Support | Reduce Runtime Profiling Overhead |
| Fast Interpretation Support | Fast Interpretation |

Table 5.2: Hardware Acceleration Support

To show the flexibility of our acceleration scheme, we integrated a set of hardware supports and dynamic optimization techniques into our microarchitecture ISA and uop runtime for accelerating Dalvik bytecode execution, as listed in Table 5.2 on the preceding page. We implemented the atomic regions [DGB+03, NRS+07, PTBC00, NDZ10] for supporting the speculative optimizations including the control speculation [NRS+07, PTBC00] and data speculation [WWRP12] in our just-in-time compilation of the Dalvik bytecode. In case of mis-speculation, the atomic region execution will be rolled back to run the non-speculative code. For supporting control speculation, we implemented assert [PTBC00]. This allows efficient null-pointer checks, array-bound checks, etc. as the check failure rarely happens [WCW13]. For supporting data speculation, we implemented hardware memory alias detection [WWRP12], which is important for our efficient just-in-time compilation due to the large runtime overheads in the aggressive compiler memory alias analysis. We implemented a fast internal microarchitecture trap using reserved microarchitecture registers [Kre03] without the need of context switching in trap handling, which is crucial for reducing the mis-speculation trap overheads in our aggressive control/data speculations. For efficient register allocation, we implemented a large set of general purpose microarchitecture registers (64 INT register and 64 FP registers) as compared to the small set of registers in x86 (Android runs in 32-bit x86 with only 8 INT registers and 8 SSE registers). This helps a lot for our just-in-time compilation as it cannot use expensive register allocation algorithms to reduce register spills due to the large runtime overheads. We implemented hardware control transfer support [KS03, WW13] in order to reduce the translation lookup overhead for indirect branches. We implemented hardware support for profiling [Kre03] to reduce the runtime profiling overhead in our just-in-time compilation. Finally, we also implemented various hardware support for fast interpretation [Kre03, WCW13] in order to reduce the program startup overhead due to interpretation. All of these hardware supports are difficult to be supported at x86 ISA level, although we believe they can be efficiently supported at the microarchitecture level using our acceleration scheme.

Besides that, there are several natural benefits in the Dalvik bytecode execution through our uop runtime as compared to the baseline regular managed runtime. First, the regular managed runtime compiles the RISC-like Dalvik bytecode to CISC-like x86 code, which is then decoded to the RISC-like microarchitecture ISA code for pipeline execution. That brings big inefficiency to the finally generated code due to the conversion between RISC-like instructions and CISC-like instructions. Previous work [HS04] has shown that decoding x86 instructions into a number of RISC-like micro-operations tends to increase the number of operations that must be individually issued in order to execute the original program. By directly compiling the RISC-like Dalvik bytecode to RISC-like microarchitecture ISA code, our uop runtime generates much more efficient code than the baseline. Second, our uop runtime leverages detailed internal microarchitecture knowledge for efficient code generation. For example, to reduce the energy and die area, the x86 ISA typically only supports the efficient out-of-order execution of a subset of x86 floating-point instructions and runs the rest in a slow path. Our uop runtime can thus generate more efficient floating-point code with this knowledge.

**Bytecode Coverage**

We created the checkpoint for two 0xBench benchmarks, linpack and scimark. Using Intel VTune Amplifier for Android SW on a XOLO PR4 SDV, we evaluated the ratio between native-code and bytecode execution. In AndEBench-Java 67.59% of all retired instructions are spent on byte-code, which roughly matched the previous measurements in our translation. In 0xBench-Linpack, it is 60% and in 0xBench-Scimark it is 91.01%. As the acceleration technology is working on bytecode it is only speeding-up this portion of the benchmark - thus the acceleration performance results achieved on AndEBench are excellent. We expect results to be similar on 0xBench-Linpack and even better on 0xBench-Scimark.

### Workloads

We use three suites of Android benchmarks for evaluating our acceleration scheme: CaffeineMark, AndEBench and 0xBench. The CaffeineMark is a series of 9 tests designed to measure various aspects of the Java virtual machine (VM) performance on various hardware and software configurations [Cor97]. AndEBench is a suite of benchmarks developed by the Embedded Microprocessor Benchmark Consortium (EEMBC) for Android devices [Con12]. It provides a standardized, industry-accepted method of evaluating Android platform performance. 0xBench is an Android benchmark suite developed by 0xlab [0xl11]. It integrates a series of benchmarks for the Android system into a comprehensive benchmark suite.

To compare Dalvik bytecode execution through our micro-virtual machine and through the regular virtual machine, we put special markers in the program in order to measure the performance and energy consumption for the same bytecode snippets. For each benchmark program, we collected data on the execution of 25 randomly selected snippets and average over them to report the data for that program. For each measurement, we run at least 1 billion instructions to warm up the just-in-time compilation in the functional execution of the simulator, then at least 100 million instructions to warm up the simulator microarchitecture states, and at last collect performance and energy data on the execution of at least 300 million instructions between markers.

Most of our speedup comes from the reduction of the microarchitecture ISA instructions in our just-in-time compilation. We classified all of the microarchitecture ISA instructions into 4 categories: ALU, BRANCH, LOAD and STORE. Figure 5.4 shows the dynamic instruction reduction for the different categories. Overall, we reduce 42% ALU instructions, 26% branch instructions, 16% load instructions and 28% store instructions. Most of the branch instructions are reduced due to the control speculation on null-pointer check and array-bound check with asserts [PTBC00]. The load/store instructions reductions are due to the large microarchitecture ISA register set and the data speculation support in the microarchitecture ISA.

## 5.7 Evaluation

### Performance

We used the same Caffeinemark benchmark we presented in Section 3.4.4 on page 51, which consists of six benchmarks: logic, loop, method, sieve, string and float. We excluded the string test which mainly evaluates the garbage collection performance (more than 90% of the time is spent in garbage collection), and which cannot leverage our acceleration scheme for speedup. So we excluded it from our measurement. The logic program mainly evaluates the conditional branch execution. The loop program mainly evaluates loop execution. The method program mainly evaluates method call and return. The sieve mainly evaluates integer division operation and the float program mainly evaluates floating point operations. Figure 5.3 shows the CaffeineMark speedup through our acceleration. Overall, our acceleration speeds up the programs by 45%. We get a big speedup for float. This is because, for energy efficiency, the microarchitecture ISA is designed to run efficiently only on a subset of x86 floating-point instructions. Our just-in-time compilation in the micro-virtual machine generates the most efficient microarchitecture ISA code for floating-point operations, while the just-in-time compilation in the regular virtual machine does not have the knowledge necessary to generate efficient code at the x86 level.

Most of our speedup comes from the reduction of the microarchitecture ISA instructions in our just-in-time compilation. We classify all the microarchitecture ISA instructions into 4 categories: ALU, BRANCH, LOAD and STORE. Figure 5.4 shows the dynamic instruc-

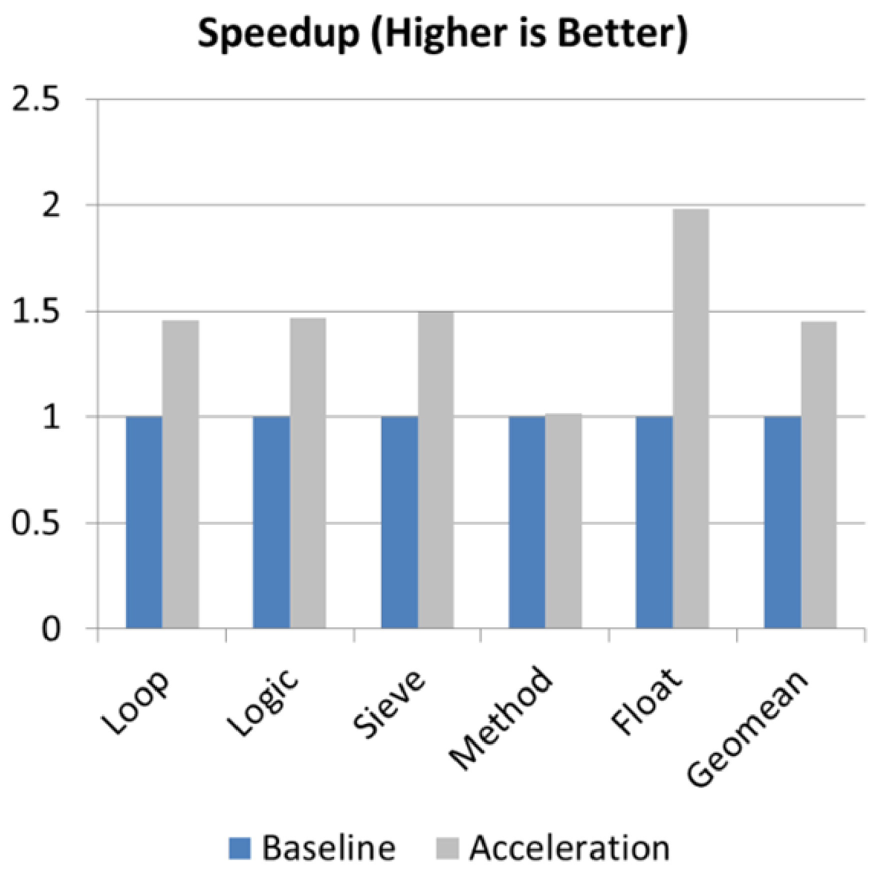


Figure 5.3: CaffeineMark Performance

tion reduction for different categories. Overall, we reduce 42% ALU instructions, 26% branch instructions, 16% load instructions, and 28% store instructions. Many branch instructions are eliminated due to the control speculation on null-pointer checks and array-bound checks with asserts [PTBC00]. The load/store instructions reductions are due to load/store elimination optimization leveraging the large microarchitecture ISA register file and the data speculation support in the microarchitecture ISA.

For *method*, our acceleration achieved big instruction count reduction (see Figure 5.4), but no performance improvement. We traced this to the allocation stalls in the out-of-order pipeline execution as shown in Figure 5.5 on page 106. Each bar in the figure shows the allocation stalls in the pipeline execution due to the blocking in different hardware buffers: LB (load buffer, 32 entries), SB (store buffer, 24 entries) and others (reorder

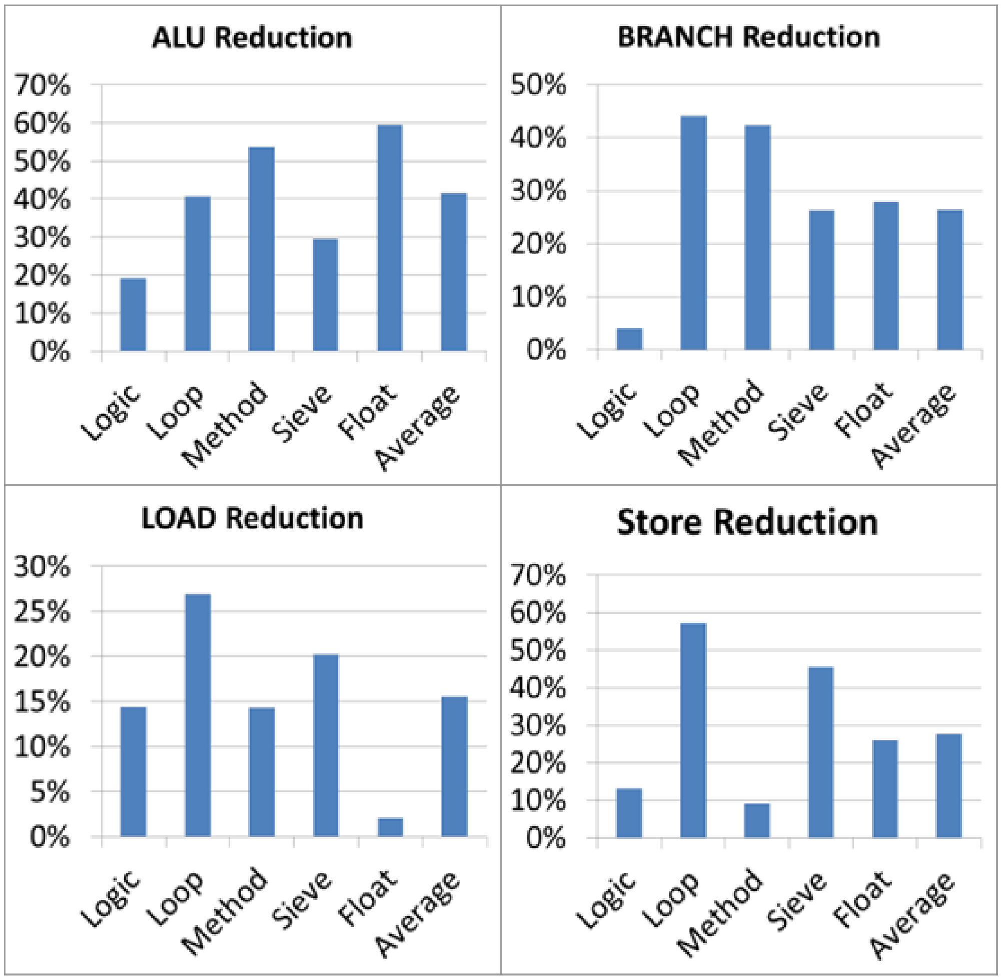


Figure 5.4: Instruction Reduction

buffer, reservation station, etc.). We can see that the method execution with our acceleration suffers from allocation stalls due to the load/store buffer overflow. Even though our acceleration reduces a lot of microarchitecture ISA instructions, the ratio of load/store instructions over other instructions increases, and thus the critical path is on the LB/SB allocation stalls. Enlarging the load/store buffer sizes may further improve the performance for our acceleration, although it will provide little help to the baseline execution. Float also suffers from high allocations stalls due to the load buffer overflow. However, the acceleration not only reduces 60% of the ALU instruction (see Figure 5.4), but also generates more efficient code, which leads to the big speedup.

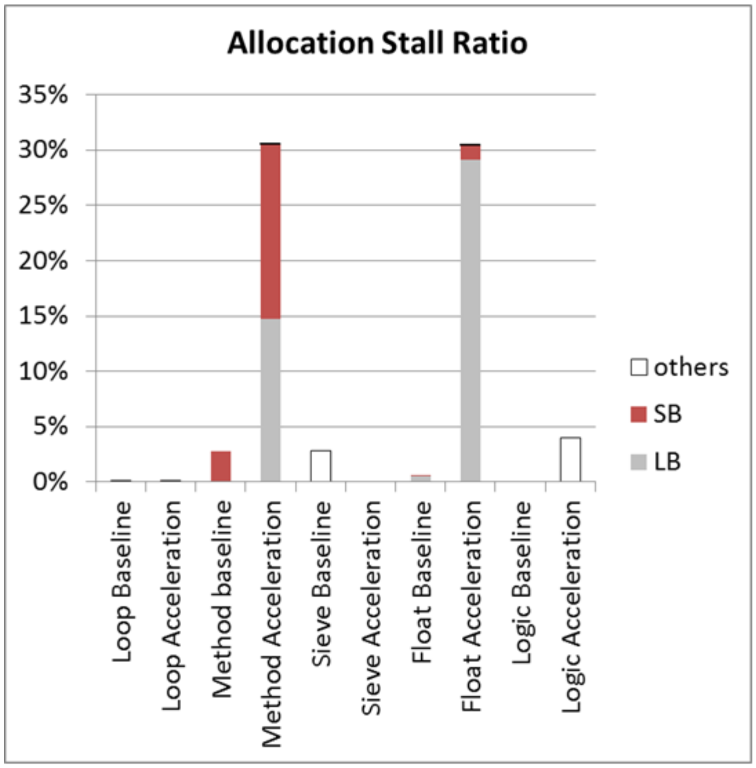


Figure 5.5: Allocation Stalls

Figure 5.6 on the following page shows the speedup on AndEBench and 0xBench benchmark suites. The 0xBench contains applications evaluating JavaScript, 2D/3D graphics, etc., instead of Dalvik bytecode. So we only evaluated our acceleration scheme on the two Dalvik benchmarks for arithmetic computing, Linpack and Scimark2. Overall, we got 23% speedup for AndEBench. AndEBench only spends 70% of its time in the Dalvik bytecode execution. The remaining 30% of time is spent in x86 native code execution, which does not run through our acceleration. So for the bytecode only, our acceleration actually improves the performance by 36%. For 0xBench, we got 80% speedup for Linpack and 48% speedup for Scimark2, with an average speedup of 64%.

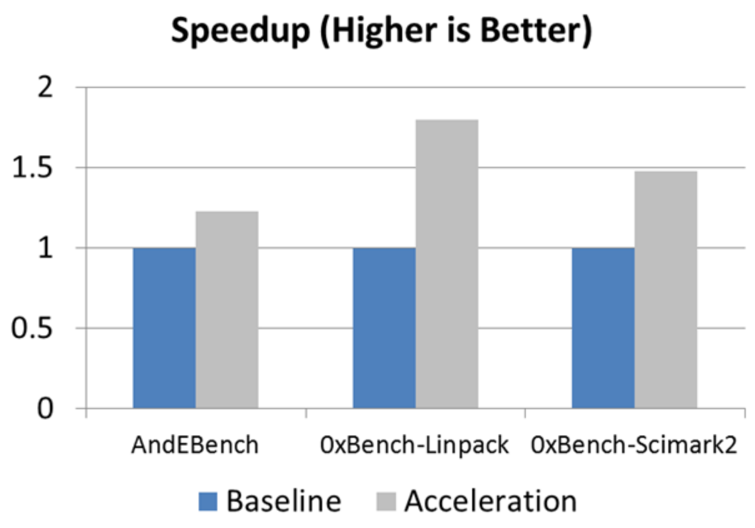


Figure 5.6: AndEBench and 0xBench Performance

### 5.7.1 Energy Consumption

We also measured the energy consumption in our simulator execution. Figure Figure 5.7 on the next page shows the normalized CPU dynamic capacitance (Cdyn), which measures the dynamic energy dissipation, in the CaffeineMark execution. We can see that our acceleration increases little to the dynamic capacitance; hence the energy consumption. This is because our performance improvement mainly comes from the instruction reductions, without affecting the energy consumption. The Cdyn for method actually gets reduced due to the frequent allocation stalls (see Figure 5.5 on the preceding page). Float has more frequent ALU activities with our acceleration, which increases the Cdyn slightly. Figure 5.8 on page 109 shows the normalized Cdyn for AndEBench and 0xBench.

### 5.7.2 Startup Overhead Reduction

One issue with the just-in-time compilation for bytecode is the program startup overheads, which impacts the user experience in interactive mobile and web applications. In previous sections, we mainly evaluated performance for the benchmarks after they warm up from

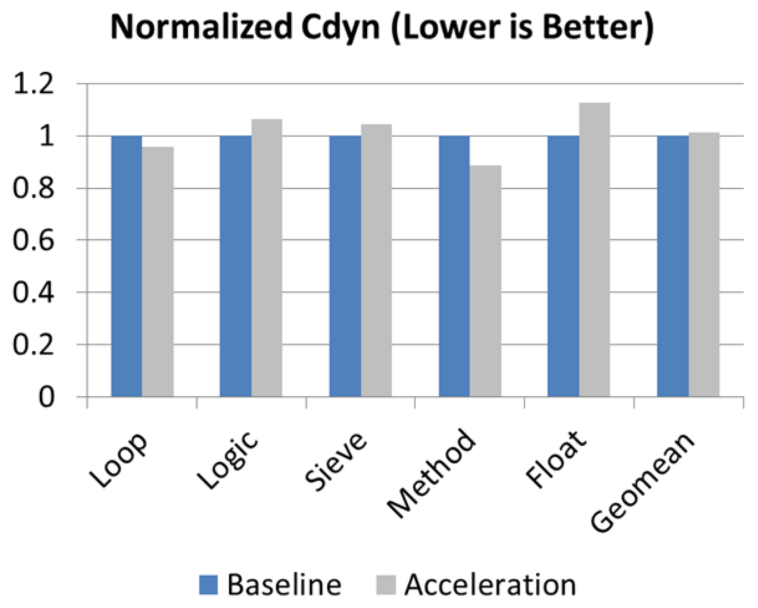


Figure 5.7: CaffeineMark Energy Consumption

just-in-time compilation. Our microarchitecture ISA also provides hardware supports such as runtime profiling support, fast interpretation support, etc., which greatly reduce the program startup overheads. Figure 5.9 on page 110 shows, from the program start, the number of microarchitecture ISA instructions executed for the corresponding number of Dalvik bytecode instructions, averaged over all of the programs in CaffeineMark. So to run the initial 1M, 10M and 100M bytecode instructions, the baseline needs to run 140M, 410M and 790M microarchitecture ISA instructions, respectively. However, our acceleration only needs to run 20M, 80M and 230M microarchitecture ISA instructions, respectively. That represents 7x, 5x and 3.4x speedups for running the initial 1M, 10M and 100M bytecode in terms of microarchitecture ISA instruction counts.

## 5.8 Conclusion

In this chapter, we present a novel acceleration scheme to permit the JIT compiler of a virtual machine to directly convert bytecodes into the internal CPU ISA, which allows flex-

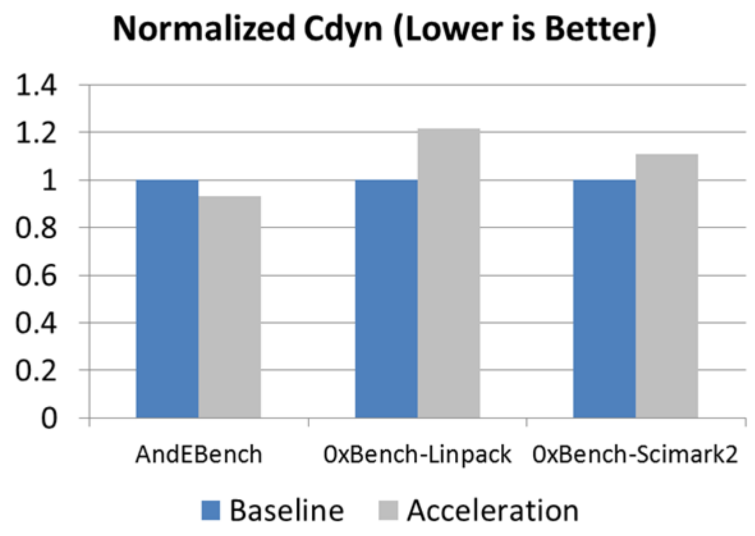


Figure 5.8: AndEBench and 0xBench Energy Consumption

ible and efficient acceleration of the general-purpose bytecode execution in a state-of-art out-of-order CPU design. Our experiments, using industry standard and scientific workloads, show significant evidence that the direct translation of bytecode into the internal ISA of an out-of-order processor significantly improved the performance of bytecode applications, without increases in energy consumption. As a future work, the acceleration interface could be extended to support JavaScript code and other dynamic typed languages, in order to demonstrate that our acceleration scheme can be deployed to accelerate other managed languages.

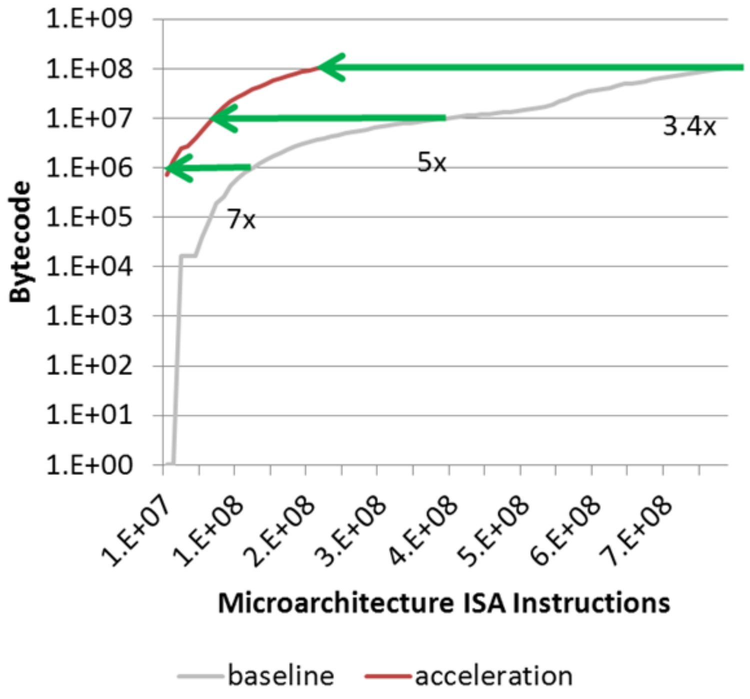


Figure 5.9: Startup Overhead Reduction

# Chapter 6

# Future Work

*"We can only see a short distance ahead, but*
*we can see plenty there that needs to be done."*
Alan Turing

In the preceding chapters we have touched on many areas where more study is required. In this chapter we present areas of future research that are more long-term in nature.

## 6.1 Promotion of Stack Variables to Registers

In this dissertation we proposed a scheme to accelerate bytecode execution. We believe more study must be done to investigate new optimizations opportunities on HW/SW co-designed systems. One future research goal is to exploit the similarity between register-based bytecode, such as the Dalvik bytecode, and the underlying register-based architectures commonly found in modern processor designs, promoting stack variables to physical registers, which can significantly enhance the performance of both interpreted and JIT translated code. Promoting stack variables at the bytecode level would avoid the problem of dealing with memory aliases between promoted stack variables and other implicit memory references that exist when stack promotion is performed in the architectural ISA code, as described in [LWH11a].

The novelty of this solution lies in the promotion of stack variables to physical registers both in interpreter execution and in the JITed code, by using the same calling convention. If a method has more local variables than the maximum number of physical registers for automatic promotion, all of its variables that cannot be promoted are stored in (spilled to) memory, and additional load/store instructions are required prior to each use or definition. Our experiments show that this is not the common case, since most methods have less than ten (10) local variables.

In Dalvik, for example, all the Non-VM related instructions (like binary operations) can benefit from stack variable promotion, since they operate on virtual registers. This shows that this technique would not be restricted to a small subset of bytecode instructions, and could provide substantial performance improvements. In one possible implementation, up to n registers could be reserved for stack variables promotion. We believe $n = 10$ physical registers is enough for well-designed methods of modern object oriented languages. This future work can unlock further optimization opportunities in co-designed processors, for example the exec instruction available in the Transmeta Efficeon [DGB+03], which allows dynamic modification of the operands of an instruction. For example, to execute an add v0, v1, v2 instruction, the interpreter would first decode the bytecode instruction, identifying the local variables involved, and then it would dynamically produce an instruction to add the corresponding physical registers, or to load local variables from memory in case they have not been promoted, which would be rare. Contrary to previous solutions, this technique would not require modifications to the internal intermediate representation, nor additional translation of the intermediate representation.

```
var_type_map = new hash_map(method.max_local_variables);
for each bytecode b of method M do
    switch (bytecode.op) {
       if (bytecode defines a double/long variable dst) {
           var_type_map.add(dst, dword);
       } else {
           var_type_map.add(dst, sword);
       }
       if ( bytecode operates on double/long src operands) {
          opnd_type = dword;
       } else {
          opnd_type = sword;
       }
       for each operand opnd in bytecode do
              var_type_map.add(src, opnd_type);
       }
}
```

Listing 6.1: Algorithm to detect at runtime the data type of stack variables

We use the Dalvik VM to illustrate the stack variable promotion technique, since Dalvik is a register-based VM. However, the techniques we propose can be applied to any register-based VM. In order to promote stack variables into physical registers, a single-pass algorithm has to scan the whole method bytecode, retrieving the data type of each stack variable from the semantics of each bytecode. The metadata for the method, contained in a dex file, provides the number of local variables in the method stack frame. Our algorithm uses this information to create a new data structure to map local variables into its data type. The pseudo code for the algorithm is shown in Listing 6.1 below.

Listing 6.1 processes each bytecode, creating an identity map var_type_map, which maps local variables to its data type. The mapping with the data types of the variables is used by the interpreter and JIT compiler to promote stack variables into physical registers. The single word variables (sword) are promoted to 32 bit registers. The double word (dword) variables are promoted to 64 bit physical registers.

After detecting the data types for the local variables, a single calling convention has to be defined, such that both the interpreter and JIT compiler must follow. We assume that the underlining architecture has only a set of n registers (n = 10) available for promoting local variables into registers. So, whenever there is a function call, we need to explicitly define the actions that both the caller and the callee must execute in order to preserve the physical registers allocated for register promotion.

To support method invocation, the following calling convention between caller and callee could be used:

1) **At caller**: Prior to the method call, the caller must save all local variables promoted to physical registers back to the stack frame, in memory. These include the physical registers assigned to local virtual registers and argument registers. The amount of input virtual registers in callee can be acquired from the invoke DEX instruction, so the physical argument registers in callee can be determined by the caller according to the register mapping algorithm described previously. The invoke instruction will adjust the stack for the new method.

2) **At callee entry**: Callee method does not need to save any register, since it is the responsibility of the caller to do so.

3) **Caller, After Call**: After callee returns, the caller has to reload all promoted virtual variables from stack into physical registers.

The reason the caller spills all promoted virtual registers to stack is to preserve the method invocation stack, which is used by the exception handling and the garbage collection algorithms.

Besides the performance improvements in cold code executed by the interpreter, this technique also unlocks the potential for further performance improvements on JIT translated code. Modern JIT compilers for mobile devices (ex. Dalvik JIT) usually employ a lightweight trace compiler, due to its simplicity, lower compilation overhead and speedup similar to heavy-weight JIT compilers [GPF06b]. A trace is a linear sequence of frequently executed operations. A trace contains a single entry and multiple exits, without control flow merge points. One common optimization performed by a trace compiler is the promotion of local variables into physical registers. The compiler keeps track of variables loaded into registers, reusing the physical register whenever possible. Before taking a trace side exit, and jumping back to the interpreter, all promoted registers are spilled back to memory. Whenever a side exit of a trace becomes hot, a new trace is formed, and again, local variables have to be reloaded from memory when they are first used, since each trace is compiled separately. The stack variable promotion technique avoids the loading of the variables at the beginning of a trace, as well as the store back (spill) to memory before a side exit is taken, which significantly reduces the overhead. Since most traces belong to hot code, this technique can significantly reduce the number of load store operations, and so increase the performance and energy efficiency of JIT generated code.

## 6.2 Universal Acceleration Bytecode

As a generalization of our framework, we envision the design of a universal *acceleration bytecode* format. Such an *acceleration bytecode* would be the interface to accelerate applications written in different languages. This universal format would have special bytecode instructions, for example to target the dynamic typed languages such as PHP and JavaScript. This would allow the bytecode accelerator to leverage the higher level semantics of dynamic typed languages and perform aggressive speculative optimizations, such as type specialization, to improve the performance per watt of these applications.

## 6.3 Fill the Dark Silicon with Specialized cores for Bytecode Acceleration

Dark silicon has emerged as the fundamental limiter in modern processor designs. The GreenDroid [GHSV+11] is a research mobile processor prototype that provides specialized energy-reducing cores targeting key portions of Google's Android smartphone platforms, which is an effective approach to use the dark silicon to execute general-purpose smartphone applications with 11 times less energy use than today's most efficient designs. GreenDroid does not directly accelerate bytecode execution. Instead, it designs special c-cores for accelerating the frequently-executed code in the Android runtime library and in the Dalvik virtual machine.

As a future work, the bytecode acceleration framework proposed in this dissertation can be extended to dynamically translate common fragments of bytecode to run natively in the conservation cores, similar to the approach used in the GreenDroid. This would allow filling the chip's dark silicon area with specialized conservation cores in order to save energy on common applications running atop the mobile platform. We believe the use of special hardware acceleration can significantly improve the energy efficiency of special purpose applications running in resource constrained devices, such as the emerging wearable devices, which have very popular in recent years.

## 6.4 HW/SW Codesigned Security

Due to the extensive growth and diffusion of mobile devices, and to their increasingly powerful capabilities, many users are massively using mobile devices both for personal and work related activities. As such, these devices can expose personal and organizational data if not properly protected. This poses serious threats to the security of such devices.

The low power budget and reduced computational capabilities of such kinds of devices impose several constraints that must be fulfilled in order to provide effective and suitable security schemes. We envision our framework being used to enable the development of more sophisticated security mechanisms, along with the development of innovative security hardware support, with little or no increase in the energy consumption and performance requirements, which could provide a more secure environment for applications running in mobile devices.

# Chapter 7

# Conclusion

*"I think and think for months and years.*
*Ninety-nine times, the conclusion is false.*
*The hundredth time I am right."*
Albert Einstein

This concluding chapter discusses what we have accomplished in this dissertation.

## 7.1 Discussion

Engineering involves the analysis of tradeoffs. In the case of mobile processor design this analysis is particularly complex and difficult to understand completely as these processors have stringent constraints on power, minimized die area, and must deliver outstanding performance for today's coolest applications, fast web browsing, and seamless connectivity. Unfortunately, these characteristics are often adversarial, and improving one often results in worsening the others. This dissertation presented dynamic optimization techniques that improve both the performance and energy efficiency of mobile processors.

Ever since the first stored program computer was designed 60 years ago, microarchitecture and compiler technology have both evolved tremendously. These advances encompass: high speed processors and memory, microarchitecture support for speculative execution, advances in compiler techniques for data and control analysis, dynamic compilation, dynamic binary translation, among others, which allowed the development of innovative computer systems technologies. However, for decades, processor designers have strived to maintain ISA compatibility, usually hiding rich hardware features available in the CPU, in order to fulfill the requirement of running existing applications on new processor designs.

HW/SW Co-Design technologies have emerged as an alternative to overcome the ISA compatibility issue, and permit to architect innovative implementation ISAs and microarchitecture support to fulfill the performance and energy efficiency requirements of modern mobile devices. In this dissertation we demonstrated that modern mobile software applications can greatly benefit from the feature-rich implementation ISA of modern processors, which is the real interface that the hardware pipeline implements, and can be designed with relatively more freedom to realize architecture innovations to fulfill the performance and/or energy efficiency requirements.

This dissertation tackles the problem of developing new Just-in-Time compilation techniques for bytecode acceleration by exploiting the hardware features of modern codesigned processors, which are not exposed at the architectural ISA in order to preserve ISA compatibility. We have established in this thesis statement that: *This thesis explores the effects, on performance and energy efficiency, of optimizations that are made possible when a JIT can directly compile bytecodes to the internal, flexibly-designed, ISA rather than the external, architectural ISA of a microprocessor.*. To support our claim, we provided the design of an acceleration framework to accelerate the execution of applications written in modern object oriented high level languages, such as Java and C#, which are compiled into a platform independent ISA, with richer semantics compared to the architectural ISA of existing systems. Our acceleration framework allows the JIT compiler of a managed runtime to bypass the architectural ISA, and directly translates bytecodes into the implementation ISA of modern co-designed processors, and performs code optimizations on-the-fly, to exploit the existing hardware features at the micro-architecture level. In our experiments, we observed that the additional registers and the support for speculative execution at the micro-architecture level, are the two hardware features that can deliver the best benefits in terms of performance and energy efficiency. Even though we used the x86 ISA, we believe the same ISA compatibility limitation in other architectural ISA implementations, such as ARM and SPARC, is also a major roadblock to make the best use of the underlying hardware.

As a supporting evidence, we provided two implementations of our bytecode acceleration framework, the first on a real co-designed machine, and the second on a cycle accurate simulator for a modern co-designed processor, both showing solid performance improvements with no increases in energy consumption. Our results is a significant evidence that codesigned systems combined with modern High Level Languages VM abstractions is a viable approach to combine the advances in hardware and software to deliver new heights of energy performance efficiency. Our acceleration framework demonstrated that the ISA compatibility requirement of modern mobile processor designs is a major roadblock to make the best use of the underlying hardware facilities. This is because several innovations at the microarchitecture level cannot be easily exploited by a compiler, as we have shown in this dissertation.

We showed that our transparent interface has other applications other than bytecode acceleration by implementing a simulation control interface that requires reduced implementation efforts to control the simulation of bytecode applications across multiple levels of the application software stack, facilitating the validation as well as the performance evaluation of processor designs.

We believe that our transparent interface, based on a OS driver, is an effective approach to introduce new techniques in computer architecture while providing backwards binary compatibility. As such, this thesis provides the groundwork for research on specialized in-core hardware accelerators, with the potential to expand the capabilities of mobile devices and provide a richer experience to the user. Furthermore, it opens new avenues for future research on codesigned optimizations, both to address the aforementioned energy and performance requirements, as well as to produce low cost mobile processor designs, which has become a critical design requirement as high-end features are embedded on low cost mobile devices.

We achieved our goals of making AccelDroid deployable, efficient, transparent, practical, and modular, so that it can easily be added to an existing Virtual Machine, requiring a lot less system programming effort compared to existing hardware accelerators. It took a significant amount of effort to build the system, with many design choices along the way. Even though each decision required novel research, we did not have time for exhaustive exploration of every design point. We hope that others will benefit from our framework, from the lessons we learned, and continue the research on innovative co-designed JIT compilation techniques to enable the design of next generation processors to fulfill the growing demands of modern mobile devices for more performance and energy efficiency.

## 7.2 Limitations

Our acceleration framework, just like any software system of significant complexity that exists in a large design space, inevitably has tradeoffs, limitations, and pitfalls. This section discusses some of the limitations of AccelDroid. First, as a new acceleration interface, AccelDroid has not been fully stressed by the demands of large software applications running in mobile devices. Second, probably the biggest limitation of AccelDroid is that a lot of applications running on Android platforms are shipped with significant amounts of native code. AccelDroid cannot accelerate applications on which most of the code is written in the native architectural ISA. Lastly, if the application execution is constantly alternating between bytecode accelerated execution and native code execution (code in x86 ISA), due to JNI calls or due to the execution of an OS dependent bytecode instruction (like new-array), our framework delivers less performance-per-watt improvements. This is due to the additional overhead to switch from x86 to bytecode acceleration execution mode, which reduces our gains.

## 7.3 A Final Word

The techniques proposed in this dissertation provide a groundwork for future research on HW/SW codesigned optimizations for managed runtime environments. Specifically, this study showed that by using the bytecode acceleration framework proposed in this thesis, improved performance-per-watt can be achieved for mobile applications written in modern programming languages that execute on managed runtime of mobile devices. These initial results provide significant evidence of the enormous power and viability of our approach to utilize the rich semantic of bytecodes to perform dynamic optimizations at the microarchitecture level, to benefit from the hardware support of modern, codesigned processors.

We demonstrated that our acceleration interface is an effective approach to communicate high level language abstractions to the underlying processor ISA implementation, providing the building blocks for the design of innovative hardware support and speculative optimizations to effectively deliver the stringent energy and performance requirements of modern processors. As such, our acceleration interface excels in making the best usage of the constrained resources of mobile devices, enabling the deployment of more features, without sacrificing the usability requirements, which is a critical aspect of modern software that runs on mobile devices.

Recently, since Android version 4.4, the Android runtime uses an Ahead of Time Compiler (AOT), the ART (Android RunTime), which translates the DEX bytecodes into the native ISA of the mobile phone when the application is installed. The Java Hotspot runtime also employs an AOT compilation system, which translates java bytecodes into the native ISA when a Java class (.class file) is loaded for execution. In addition, Hotspot also has a JIT compiler that re-translates hot regions of the code, during program execution, using more aggressive profile-directed optimizations. We are fairly certain that the two most fruitful future works growing out of this dissertation will focus upon: 1) extending the proposed acceleration framework to work in a Runtime environment based on AOT compilation, and 2) designing new hardware support for bytecode acceleration. Both avenues will require new design choices to tradeoff memory usage and silicon for improved performance and reduced energy consumption, which is a promising research area and an immediate continuation of our work.

# Bibliography

[0xl11] 0xlab, *0xbench: Comprehensive benchmark suite for android.*, 2011.

[ANH99] Ana Azevedo, Alex Nicolau, and Joe Hummel, *Java annotation-aware just-in-time (ajit) complilation system*, Proceedings of the ACM 1999 Conference on Java Grande (New York, NY, USA), JAVA '99, ACM, 1999, pp. 142–151.

[App06] Apple, *Apple website on rosetta. apple computers*, 2006, http://www.apple.com/rosetta/.

[ARM14] Advanced Risc Machine ARM, *Jazelle - arm*.

[aS00] aJile Systems, *aj100 real-time low power java processor, preliminary data sheet*, 2000.

[BB09] Dan Bornstein Bill Buzbee, Ben Cheng, *Dalvik jit compiler*, 2009.

[BBS$^+$00] D.M. Brooks, P. Bose, S.E. Schuster, H. Jacobson, P.N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P.W. Cook, *Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors*, Micro, IEEE **20** (2000), no. 6, 26–44.

[BCW$^+$10] Michael Bebenita, Mason Chang, Gregor Wagner, Andreas Gal, Christian Wimmer, and Michael Franz, *Trace-based compilation in execution environments without interpreters*, Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (New York, NY, USA), PPPJ '10, ACM, 2010, pp. 59–68.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia, *Dynamo: A transparent dynamic optimization system*, Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (New York, NY, USA), PLDI '00, ACM, 2000, pp. 1–12.

[BDE$^+$03] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach, *Ia-32 execution layer: A two-phase dynamic translator designed to support ia-32 applications on itanium&#174;-based systems*, Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (Washington, DC, USA), MICRO 36, IEEE Computer Society, 2003, pp. 191–.

[BGA03] Derek Bruening, Timothy Garnett, and Saman Amarasinghe, *An infrastructure for adaptive dynamic optimization*, Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (Washington, DC, USA), CGO '03, IEEE Computer Society, 2003, pp. 265–275.

[BGH$^+$06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann, *The dacapo benchmarks: Java benchmarking development and analysis*, Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (New York, NY, USA), OOPSLA '06, ACM, 2006, pp. 169–190.

[BH04] W. Binder and J. Hulaas, *A portable cpu-management framework for java*, Internet Computing, IEEE **8** (2004), no. 5, 74–83.

[BHM07] Walter Binder, Jarle Hulaas, and Philippe Moret, *Advanced java bytecode instrumentation*, Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (New York, NY, USA), PPPJ '07, ACM, 2007, pp. 135–144.

[BO11] Garo Bournoutian and Alex Orailoglu, *Dynamic, multi-core cache coherence architecture for power-sensitive mobile processors*, Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (New York, NY, USA), CODES+ISSS '11, ACM, 2011, pp. 89–98.

[BWBW11] Edson Borin, Youfeng Wu, Mauricio Breternitz, and Cheng Wang, *Lar-cc: Large atomic regions with conditional commits*, Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (Washington, DC, USA), CGO '11, IEEE Computer Society, 2011, pp. 54–63.

[BWW$^+$10] Edson Borin, Youfeng Wu, Cheng Wang, Wei Liu, Mauricio Breternitz, Jr., Shiliang Hu, Esfir Natanzon, Shai Rotem, and Roni Rosner, *Tao: Two-level atomicity for dynamic binary optimizations*, Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (New York, NY, USA), CGO '10, ACM, 2010, pp. 12–21.

[CG01] Ramon Canal and Antonio González, *Reducing the complexity of the issue logic*, Proceedings of the 15th International Conference on Supercomputing (New York, NY, USA), ICS '01, ACM, 2001, pp. 312–320.

[CHH$^+$98a] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Bharadwaj Yadavalli, and J. Yates, *Fx!32 a profile-directed binary translator*, Micro, IEEE **18** (1998), no. 2, 56–64.

[CHH+98b] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates, *Fx!32: A profile-directed binary translator*, IEEE Micro **18** (1998), no. 2, 56–64.

[Con12] Embedded Microprocessor Benchmark Consortium, *Andebench: An eembc benchmark for android devices.*, 2012.

[Cor97] Pendragon Software Corporation, *Caffeinemark benchmark 3.0*, 1997.

[Cor04] Transmeta Corporation, *Efficeon processor product brief*.

[Cor13] Intel Corporation, *Intelr advanced vector extensions 2 programming reference*, 2013.

[DGB+03] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson, *The transmeta code morphing&trade; software: Using speculation, recovery, and adaptive retranslation to address real-life challenges*, Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (Washington, DC, USA), CGO '03, IEEE Computer Society, 2003, pp. 15–24.

[Die98] Keith Diefendorff, *K7 challenges intel*, Oct. 1998, pp. 1–7.

[DPC+00] Sujit Dey, Debashis Panigrahi, Li Chen, Clark N. Taylor, Krishna Sekar, and Pablo Sanchez, *Using a soft core in a soc design: Experiences with picojava*, 2000.

[DS84] L. Peter Deutsch and Allan M. Schiffman, *Efficient implementation of the smalltalk-80 system*, Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (New York, NY, USA), POPL '84, ACM, 1984, pp. 297–302.

[EA97] Kemal Ebcioğlu and Erik R. Altman, *Daisy: Dynamic compilation for 100% architectural compatibility*, SIGARCH Comput. Archit. News **25** (1997), no. 2, 26–37.

[ecm] *ECMAScript Language Specification (Standard ECMA-262)*, Tech. report.

[ECM10] ECMA International, *Standard ecma-335 - common language infrastructure (cli)*, 5 ed., Geneva, Switzerland, December 2010.

[FBC+01] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S.J. Patel, and Steven S. Lumetta, *Performance characterization of a hardware mechanism for dynamic optimization*, Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on, Dec 2001, pp. 16–27.

[Fra94] Michael Franz, *Code-generation on-the-fly: A key to portable software*, 1994.

[Gal06] Andreas Gal, *Efficient bytecode verification and compilation in a virtual machine*, Ph.D. thesis, Irvine, CA, USA, 2006.

[GAS$^+$00] Michael Gschwind, Erik R. Altman, Sumedh Sathaye, Paul Ledak, and David Appenzeller, *Dynamic and transparent binary translation*, Computer **33** (2000), no. 3, 54–59.

[GF06] Andreas Gal and Michael Franz, *Incremental dynamic code generation with trace trees*, 2006.

[GHSV$^+$11] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor, *The greendroid mobile application processor: An architecture for silicon's dark future*, Micro, IEEE **31** (2011), no. 2, 86–95.

[Got10] Samuel S. F. Goto, *Architecture description languages synthesis*, 2010, Master's Thesis.

[GPF06a] Andreas Gal, Christian W. Probst, and Michael Franz, *Hotpathvm: An effective jit compiler for resource-constrained devices*, Proceedings of the 2Nd International Conference on Virtual Execution Environments (New York, NY, USA), VEE '06, ACM, 2006, pp. 144–153.

[GPF06b] ______, *Hotpathvm: An effective jit compiler for resource-constrained devices*, Proceedings of the 2Nd International Conference on Virtual Execution Environments (New York, NY, USA), VEE '06, ACM, 2006, pp. 144–153.

[GPF08] ______, *Java bytecode verification via static single assignment form*, ACM Trans. Program. Lang. Syst. **30** (2008), no. 4, 21:1–21:21.

[GPM$^+$99] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers, *An evaluation of staged run-time optimizations in dyc*, Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (New York, NY, USA), PLDI '99, ACM, 1999, pp. 293–304.

[Gwe95] Linley Gwennap, *Intel's p6 uses decoupled superscalar design*, 1995, pp. 9–15.

[GWW$^+$06] Bolei Guo, Youfeng Wu, Cheng Wang, Matthew J. Bridges, Guilherme Ottoni, Neil Vachharajani, Jonathan Chang, and David I. August, *Selective runtime memory disambiguation in a dynamic binary translator*, Proceedings of the 15th International Conference on Compiler Construction (Berlin, Heidelberg), CC'06, Springer-Verlag, 2006, pp. 65–79.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar, *Optimizing dynamically-typed object-oriented languages with polymorphic inline caches*, Proceedings of the European Conference on Object-Oriented Programming (London, UK, UK), ECOOP '91, Springer-Verlag, 1991, pp. 21–38.

[Hen00] John L. Henning, *Spec cpu2000: Measuring cpu performance in the new millennium*, Computer **33** (2000), no. 7, 28–35.

[HJB$^+$82] John Hennessy, Norman Jouppi, Forest Baskett, Thomas Gross, and John Gill, *Hardware/software tradeoffs for increased performance*, Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems (New York, NY, USA), ASPLOS I, ACM, 1982, pp. 2–11.

[HM11] Christian Häubl and Hanspeter Mössenböck, *Trace-based compilation for the java hotspot virtual machine*, Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (New York, NY, USA), PPPJ '11, ACM, 2011, pp. 129–138.

[HMC$^+$93] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery, *The superblock: An effective technique for vliw and superscalar compilation*, J. Supercomput. **7** (1993), no. 1-2, 229–248.

[HP03] John L. Hennessy and David A. Patterson, *Computer architecture: A quantitative approach*, 3 ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[HS97] Paul J.M. Havinga and Gerard J. M. Smit, *Low power system design techniques for mobile computers*, 1997.

[HS00] Paul J. M. Havinga and Gerard J. M. Smit, *Design techniques for low power systems*, 2000.

[HS04] Shiliang Hu and James E. Smith, *Using dynamic binary translation to fuse dependent instructions*, Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (Washington, DC, USA), CGO '04, IEEE Computer Society, 2004, pp. 213–.

[HU96] Urs Hölzle and David Ungar, *Reconciling responsiveness with performance in pure object-oriented languages*, ACM Trans. Program. Lang. Syst. **18** (1996), no. 4, 355–400.

[Int08] Intel Corporation, *Intel® Atom™Processor N270*, no. 320436-002US, September 2008.

[Int12] ______, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, no. 325462-044US, August 2012.

[KC01] Chandra Krintz and Brad Calder, *Using annotations to reduce dynamic optimization time*, Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (New York, NY, USA), PLDI '01, ACM, 2001, pp. 156–167.

[KF01] Thomas Kistler and Michael Franz, *Continuous program optimization: Design and evaluation*, IEEE Trans. Comput. **50** (2001), no. 6, 549–566.

[Kla00a] Alexander Klaiber, *The technology behind the crusoe processors*, Tech. report, January 2000.

[Kla00b] ———, *The technology behind the crusoe processors*.

[KN11] Naveen Kumar and Naveen Neelakantam, *Indirect branches in the transmeta efficeon processor.*, Proceedings of the 2011 Workshop on Infrastructure for Software/Hardware co-design (2011), Chamonix, France.

[Kre03] K. Krewell, *Transmeta gets more efficeon*, Microprocessor Report. (2003).

[KS03] Ho-Seop Kim and James E. Smith, *Hardware support for control transfers in code caches*, Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (Washington, DC, USA), MICRO 36, IEEE Computer Society, 2003, pp. 253–.

[Lau94] G. Lauterbach, *Accelerating architectural simulation by parallel execution of trace samples*, System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on, vol. 1, Jan 1994, pp. 205–210.

[LCM$^+$05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, *Pin: Building customized program analysis tools with dynamic instrumentation*, Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (New York, NY, USA), PLDI '05, ACM, 2005, pp. 190–200.

[Ler01] Xavier Leroy, *Java bytecode verification: An overview*, Proceedings of the 13th International Conference on Computer Aided Verification (London, UK, UK), CAV '01, Springer-Verlag, 2001, pp. 265–285.

[LGM$^+$14] Marc Lupon, Enric Gibert, Grigorios Magklis, Sridhar Samudrala, Raúl Martínez, Kyriakos Stavrou, and David R. Ditzel, *Speculative hardware/software co-designed floating-point multiply-add fusion*, SIGARCH Comput. Archit. News **42** (2014), no. 1, 623–638.

[LvDDM06] Han Lee, Daniel von Dincklage, Amer Diwan, and J. Eliot B. Moss, *Understanding the behavior of compiler optimizations*, Softw. Pract. Exper. **36** (2006), no. 8, 835–844.

[LWH11a] Jianjun Li, Chenggang Wu, and Wei-Chung Hsu, *Dynamic register promotion of stack variables*, Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (Washington, DC, USA), CGO '11, IEEE Computer Society, 2011, pp. 21–31.

[LWH11b] ———, *Efficient and effective misaligned data access handling in a dynamic binary translation system*, ACM Trans. Archit. Code Optim. **8** (2011), no. 2, 7:1–7:29.

[LY99] Tim Lindholm and Frank Yellin, *Java virtual machine specification*, 2nd ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Man72] Richard L. Mandell, *Hardware/software trade-offs: Reasons and directions*, Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I (New York, NY, USA), AFIPS '72 (Fall, part I), ACM, 1972, pp. 453–459.

[Mic99] Sun Microsystems, *picojava-ii programmerâĂŹs reference manual*, 1999.

[MWG00] Erik Meijer, Redmond Wa, and John Gough, *Technical overview of the common language runtime*, 2000.

[NDZ10] Naveen Neelakantam, David R. Ditzel, and Craig Zilles, *A real system evaluation of hardware atomicity for software speculation*, Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (New York, NY, USA), ASPLOS XV, ACM, 2010, pp. 29–38.

[NRS$^+$07] Naveen Neelakantam, Ravi Rajwar, Suresh Srinivas, Uma Srinivasan, and Craig Zilles, *Hardware atomicity for reliable software speculation*, Proceedings of the 34th Annual International Symposium on Computer Architecture (New York, NY, USA), ISCA '07, ACM, 2007, pp. 174–185.

[OKCM12] Hyeong-Seok Oh, Beom-Jun Kim, Hyung-Kyu Choi, and Soo-Mook Moon, *Evaluation of android dalvik virtual machine*, Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems (New York, NY, USA), JTRES '12, ACM, 2012, pp. 115–124.

[OT97] J.M. O'Connor and M. Tremblay, *picojava-i: the java virtual machine in hardware*, Micro, IEEE **17** (1997), no. 2, 45–53.

[Pat89] Yale N. Patt, *Microarchitecture choices (implementation of the vax)*, SIGMICRO Newsl. **20** (1989), no. 3, 213–216.

[PD80] David A. Patterson and David R. Ditzel, *The case for the reduced instruction set computer*, SIGARCH Comput. Archit. News **8** (1980), no. 6, 25–33.

[PHC03a] Erez Perelman, Greg Hamerly, and Brad Calder, *Picking statistically valid and early simulation points*, Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (Washington, DC, USA), PACT '03, IEEE Computer Society, 2003, pp. 244–.

[PHC03b] ———, *Picking statistically valid and early simulation points*, Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (Washington, DC, USA), PACT '03, IEEE Computer Society, 2003, pp. 244–.

[Php] Php.net, *Php5*.

[PJS97] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith, *Complexity-effective superscalar processors*, SIGARCH Comput. Archit. News **25** (1997), no. 2, 206–218.

[PL99] Sanjay J. Patel and Steven S. Lumetta, *replay: A hardware framework for dynamic program optimization*, Tech. report, IEEE Transactions on Computers, 1999.

[PLP$^+$07] E. Perelman, J. Lau, H. Patil, A. Jaleel, G. Hamerly, and B. Calder, *Cross binary simulation points*, Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on, April 2007, pp. 179–189.

[Pro01] Mark Probst, *Fast machine-adaptable dynamic binary translation*, In Proceedings of the Workshop on Binary Translation, 2001.

[PS07] Wolfgang Puffitsch and Martin Schoeberl, *picojava-ii in an fpga*, Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (New York, NY, USA), JTRES '07, ACM, 2007, pp. 213–221.

[PTBC00] Sanjay J. Patel, Tony Tung, Satarupa Bose, and Matthew M. Crum, *Increasing the size of atomic instruction blocks using control flow assertions*, Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (New York, NY, USA), MICRO 33, ACM, 2000, pp. 303–313.

[Qia00] Zhenyu Qian, *Standard fixpoint iteration for java bytecode verification*, ACM Trans. Program. Lang. Syst. **22** (2000), no. 4, 638–672.

[SAMC98] Kevin Skadron, Pritpal S. Ahuja, Margaret Martonosi, and Douglas W. Clark, *Improving prediction for procedure returns with return-address-stack repair mechanisms*, Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (Los Alamitos, CA, USA), MICRO 31, IEEE Computer Society Press, 1998, pp. 259–271.

[SCF$^+$03] Brian Slechta, David Crowe, Brian Fahs, Michael Fertig, Gregory Muthler, Justin Quek, Francesco Spadini, Sanjay J. Patel, and Steven S. Lumetta, *Dynamic optimization of micro-operations*, Proceedings of the 9th International Symposium on High-Performance Computer Architecture (Washington, DC, USA), HPCA '03, IEEE Computer Society, 2003, pp. 165–.

[SCK$^+$93] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson, *Binary translation*, Commun. ACM **36** (1993), no. 2, 69–81.

[Sla94] Michael Slater, *Amd's k5 designed to outrun pentium*, 1994.

[SN05] Jim Smith and Ravi Nair, *Virtual machines: Versatile platforms for systems and processes (the morgan kaufmann series in computer architecture and design)*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[SPC01] T. Sherwood, E. Perelman, and B. Calder, *Basic block distribution analysis to find periodic behavior and simulation points in applications*, Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on, 2001, pp. 3–14.

[SPHC02] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder, *Automatically characterizing large scale program behavior*, SIGOPS Oper. Syst. Rev. **36** (2002), no. 5, 45–57.

[SSHB98] J.E. Smith, S. Sastry, T. Heil, and T.M. Bezenek, *Achieving high performance via co-designed virtual machines*, Innovative Architecture for Future Generation High-Performance Processors and Systems, 1998, Oct 1998, pp. 77–84.

[SSNB06] Swaroop Sridhar, Jonathan S. Shapiro, Eric Northup, and Prashanth P. Bungale, *Hdtrans: An open source, low-level dynamic instrumentation system*, Proceedings of the 2Nd International Conference on Virtual Execution Environments (New York, NY, USA), VEE '06, ACM, 2006, pp. 175–185.

[SYK$^+$01] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani, *A dynamic optimization framework for a java just-in-time compiler*, SIGPLAN Not. **36** (2001), no. 11, 180–195.

[SYK$^+$05] ———, *Design and evaluation of dynamic optimizations for a java just-in-time compiler*, ACM Trans. Program. Lang. Syst. **27** (2005), no. 4, 732–785.

[SYN03] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani, *A region-based compilation technique for a java just-in-time compiler*, SIGPLAN Not. **38** (2003), no. 5, 312–323.

[Tra06] Transitive, *Transitive website - transitive corporation ltd.*, 2006, http://www.transitive.com/.

[UC00] D. Ung and C. Cifuentes, *Dynamic re-engineering of binary code with run-time feedbacks*, Reverse Engineering, 2000. Proceedings. Seventh Working Conference on, 2000, pp. 2–10.

[WCB$^+$09] Christian Wimmer, Marcelo S. Cintra, Michael Bebenita, Mason Chang, Andreas Gal, and Michael Franz, *Phase detection using trace compilation*, Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (New York, NY, USA), PPPJ '09, ACM, 2009, pp. 172–181.

[WCW13] Cheng Wang, Marcelo S. Cintra, and Youfeng Wu, *Acceldroid: Co-designed acceleration of android bytecode*, Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (Washington, DC, USA), CGO '13, IEEE Computer Society, 2013, pp. 1–10.

[Wu13] Youfeng Wu, *Hw/sw co-designed acceleration of dynamic languages*, SIGPLAN Not. **48** (2013), no. 5, 1–2.

[WW11] Cheng Wang and Youfeng Wu, *Modeling and performance evaluation of tso-preserving binary optimization*, Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (Washington, DC, USA), PACT '11, IEEE Computer Society, 2011, pp. 383–392.

[WW13] ______, *Tso_atomicity: Efficient hardware primitive for tso-preserving region optimizations*, SIGARCH Comput. Archit. News **41** (2013), no. 1, 509–520.

[WWRP12] Cheng Wang, Youfeng Wu, Hongbo Rong, and Hyunchul Park, *Smarq: Software-managed alias register queue for dynamic optimizations*, Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on, Dec 2012, pp. 425–436.

# Appendices

## A Dalvik Virtual Machine

The Dalvik Virtual Machine (DVM) is a register-based machine that performs computations using the virtual registers included in the VM. The stack frame for each method includes a register file with general purpose registers, which are mapped to local variables/arguments, as well as special registers for saving the caller method's state, such as caller's PC, the pointers to the caller's register file, and the method data, etc., which are used to restore the caller's execution state when the callee method returns.

### A.1 Dalvik Bytecodes

Dalvik bytecodes can be grouped into two categories: 1) VM-related instructions that interact with a virtual machine, and 2) computation instructions that just compute on the virtual registers. There are about 232 instructions in DEX, and 43 of them are VM-related instructions. The remaining instructions (Non-VM related) are very simple instructions such as arithmetic instructions, logic instructions, etc.

Consider the recursive method notInlinaeableSeries from class MethodT in Listing 1. The method has only one integer argument - paramInt - and one integer local variable -i-. The

```
class MethodT {
  ...
  public int notInlineableSeries(int paramInt)
  {
     int i = paramInt;
     this.depthCount += 1;
     if (paramInt == 0) {
       return i;
     }
     if ((i & 0x1) != 0) {
       i += notInlineableSeries(paramInt - 1);
     }
     else {
       i += 1 + notInlineableSeries(paramInt - 1);
     }
     return i;
  }
  ...
}
```

Listing 1: Method notInlineableSeries of class MethodT

bytecode in Listing 2 is the DEX bytecode representation for the method notInlinaeable-Series.

A standard method call, such as the one in line 11 of Listing 1 (first recursive call), will be turned into a piece of bytecode, which is often referred to as a call site. This comprises a dispatch opcode (such as invokevirtual, for regular instance method calls) and a constant (an offset into the Constant Pool of the class), which indicates which method is to be called. The bytecode at offset 000f in Listing 2 is the call site for the method call in line 11 of Listing 1. Before performing the method invocation, the class and the method identified by the method invocation bytecode (for example, invoke-virtual) are resolved. If the class file has not been loaded, the runtime will load it and initialize the proper data structures.

To invoke a method, the Dalvik VM first creates a new stack frame of the proper size, on the stack, for the new method. The stack frame contains space for the method's arguments, local variables and its operand stack. The size of the local variables and operand stack are calculated at compile-time and placed into the bytecode metadata, so the VM knows just

```
MethodT.notInlineableSeries:(I)I
|0000: iget           v0, v1, LMethodAtom;.depthCount:I
|0002: add-int/lit8   v0, v0, #int 1
|0004: iput           v0, v1, LMethodAtom;.depthCount:I
|0006: if-nez         v2, 0009
|0008: return         v2
|0009: and-int/lit8   v0, v2, #int 1
|000b: if-eqz         v0, 0016 // +000b
|000d: add-int/lit8   v0, v2, #int -1
|000f: invoke-virtual {v1, v0}, MethodT.notInlineableSeries:(I)I
|0012: move-result    v0
|0013: add-int/2addr  v0, v2
|0014: move           v2, v0
|0015: goto           0008 // -000d
|0016: add-int/lit8   v0, v2, #int -1
|0018: invoke-virtual {v1, v0}, MethodT.notInlineableSeries:(I)I
|001b: move-result    v0
|001c: add-int/lit8   v0, v0, #int 1
|001e: add-int/2addr  v0, v2
|001f: goto           0014
```
Listing 2: Dalvik bytecode for Method notInlineableSeries of class MethodT

how much memory will be needed by the method's stack frame. In Dalvik, when a method is invoked, the parameters to the method are placed into the last n registers. If a method has 2 arguments, and 5 virtual registers (v0-v4), the arguments would be placed into the last 2 registers - v3 and v4, as shown in Figure 2 on the next page.

For example, the non-static method invocation invoke-virtual v1, v0, MethodT.notInlineableSeries:(I)I calls the notInlineableSeries method passing 2 integer parameters. The first parameter is

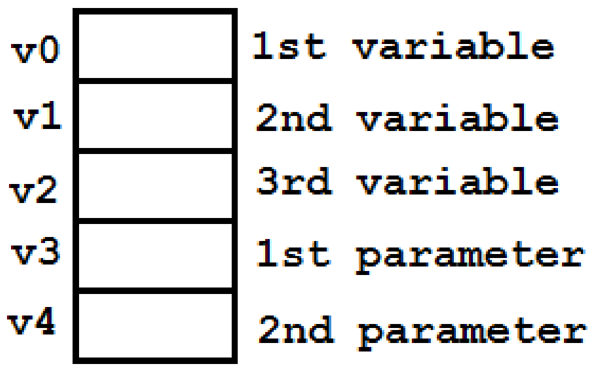


Figure 1: Stack layout of a method with 2 parameters and 3 local variables.

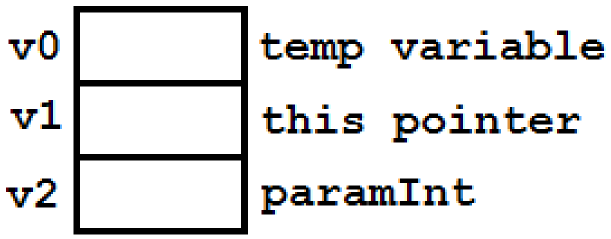


Figure 2: Stack layout of a method nonInlineableSeries.

```
public class FloatTest {
    public long twoArguments(int fpi, double fpd) {
        int fli;
        long fld;

        if (fpi > 0) {
            fli = 1;
            fld = ((long)fpd) >> 1;
        } else {
            fli = 2;
            fld = ((long)fpd) << 1;
        }

        return (fld + (long) fli);
    }
}
```

Listing 3: A method with local variables of types double and long

the implicit (method's object) pointer. The first parameter is assigned to virtual register v1, in the call stack, as shown in Figure 2. The second parameter is the integer paramInt, which is assigned to virtual register v2 in the stack frame, as shown in Figure 2.

The bytecode at offset 000f in Listing 2 invokes the method notInlineableSeries. It passes two arguments, v1 and v0, which are the first argument and the class object pointer.

The example in Figure 1 on page 135 operates only on arguments and local variables of type integer. In Dalvik bytecode registers are always 32 bits, and can hold any type of value. Two consecutive registers are used to hold 64 bit types (Long and Double). Just by looking at the method metadata it is not possible for the VM to infer what the data type of a local variable is. Listing 3 shows a method with long and double variables.

```
FloatTest.twoArguments:(ID)J
|0000: const/4          v2, #int 1
|0001: if-lez           v5, 0008
|0003: double-to-long   v0, v6
|0004: shr-long/2addr   v0, v2
|0005: int-to-long      v2, v2
|0006: add-long/2addr   v0, v2
|0007: return-wide      v0
|0008: const/4          v3, #int 2
|0009: double-to-long   v0, v6
|000a: shl-long/2addr   v0, v2
|000b: move             v2, v3
|000c: goto             0005
```

Listing 4: A method with local variables of types double and long

Listing 4 shows the resulting DEX bytecode for the function twoArguments of class FloatTest, shown in Listing 3.

## A.2 Dalvik Interpreter

The Dalvik runtime includes three interpreters, labeled “portable”, “debug”, and “fast”. The portable interpreter is largely contained within a single C function, and should compile on any system that supports a C compiler. The fast interpreter uses hand-coded assembly fragments. If none are available for the current architecture, the build system will create an interpreter out of C “stubs”. The resulting “all stubs” interpreter is quite a bit slower than the portable interpreter, making “fast” something of a misnomer. The fast interpreter is enabled by default. The “debug” interpreter is a variation of the portable interpreter that includes support for debugging and trace formation. This interpreter is used to record an execution trace when the trace-based JIT compiler is enabled. When a debugger attaches, or the tracing mode is enabled, the VM will switch interpreters at a convenient point. This is done at the same time as the GC safe point check: on a backward branch, a method return, or an exception throw. Similarly, when the debugger detaches or trace recording is discontinued, execution transfers back to the “fast” or “portable” interpreter.

We used the portable debug interpreter to implement the bytecode accelerator we describe in  Chapter 3 on page 29, and  Chapter 5 on page 85. The portable interpreter was included in a new module, integrating the DBT bytecode runtime as illustrated in  Figure 3.4 on page 38. The interpreter was extended to benefit from the support for fast interpretation, by using the LINKPIPE feature, as we will detail in  Section 3.3.1 on page 39.

## A.3 Trace Based Just-in-Time compiler

When a new application is started, the DVM starts the bytecode execution using the interpreter. A counter is maintained for the target of each jump instruction. Once the counter reaches a certain threshold, the runtime will switch the interpreter execution to the debug interpreter, which will execute the bytecode instruction and at the same time record it in the trace buffer. The DVM employs a trace-based JIT compiler [OKCM12], which uses a trace as the unit of compilation.

The trace recording is stopped when a branch or a function call is interpreted or when the number of bytecode instruction in the trace buffer exceeds a pre-defined limit (100 in our case). After a trace is formed, it is passed to a compiler thread that will compile and install it in the code cache. Dalvik also employs a runtime optimization that allows compiled traces to be chained, in order to reduce the lookup overhead every time the execution exits a compiled trace.

# B Tramseta Efficeon Processor

The low-power x86 compatible Efficeon processor [Cor04] is Transmeta's second-generation 256-bit Very Long Instruction Word (VLIW) design aimed at ultra-portable and mainstream notebook computers, as well as new classes of devices such as tablet PCs, ultra-personal computers, silent desktop computers, blade servers, thin clients and embedded systems.

The Efficeon processor has a microarchitecture designed for simplicity by moving complex but infrequent tasks into the software. Although a full discussion of the architecture is beyond the scope of this dissertation, we provide some details here that are relevant to the topic we will address in next chapters.

The Efficeon is an in-order VLIW processor, which is designed to provide high performance execution of software-scheduled code. For design simplicity and reduced power, the microarchitecture does not provide hardware interlocks or register scoreboarding, and therefore relies on a compiler to correctly schedule dependent and independent operations. To simplify the compiler's task, and to enable speculative execution, Efficeon provides hardware support for taking fast register and memory checkpoints and for reordering memory operations.

Efficeon employs a software engine, the Code Morphing Software (aka CMS) [DGB$^+$03], to convert code written for x86 processors to the native VLIW instruction set of the chip, as we describe in section Section B.3 on page 144. Like its predecessor, the Transmeta Crusoe (a 128-bit VLIW architecture), Efficeon stresses computational efficiency, low power consumption and a low thermal footprint. To maximize performance and responsiveness, the Efficeon processor features a state-of-the-art 256-bit-wide VLIW engine that can issue up to 8 instructions per clock cycle, a large 1MB L2 cache and support for SSE & SSE2 instructions, all of which help make for a compelling multimedia experience.

According to the design briefings [Cor04] released by Transmeta, the Efficeon processor has the following features:

- Advanced 256-bit VLIW engine with two load/store units, two integer ALU units, two multimedia SSE/SSE2/MMX units, a floating point unit, and a branch unit. The VLIW core can execute one 256-bit VLIW instruction per cycle, which is called a *molecule*. Each molecule has room to store eight 32-bit instructions, called *atoms*. Up to eight atoms (instructions) can be issued per cycle.
- It has 64 general-purpose registers and 32 floating point registers, allowing the architectural x86 registers to be assigned to dedicated native VLIW registers, with an ample set available for use by CMS.
- 85 million 90nm transistors on a 65mm$^2$ die area, as Figure 3 on the following page illustrates.
- It provides an advanced Code Morphing$^{\text{TM}}$ software x86-compatible for a mobile platform solution.
- It was built in a 0.13$\mu$m fabrication technology for GHz performance at very low power levels.
- It has standard product speeds of 900 MHz - 1.1 GHz at low power consumption.
- 128 KByte L1 instruction cache, 64 KByte L1 data cache, and 1024 KByte (TM8600) L2 write-back cache for high performance.
- Contains SSE, SSE2, and MMX instruction support for cinematic multimedia performance.

Efficeon CMS reserves a small portion of main memory (typically 32 MB) for its translation cache of dynamically translated x86 instructions.

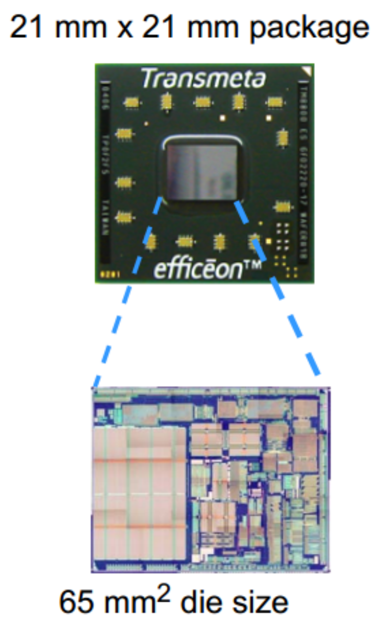


Figure 3: Efficeon Processor Package and Die Area

## B.1 Hardware Support for Speculative Execution

For high performance, the Efficeon processor has hardware support for speculative execution. Each architecture register (x86) has two copies: a shadowed and a working copy. Likewise, each data cache line has a bit to mark all speculative memory updates.

The Efficeon processor exposes its support for fast hardware checkpoints through the two operations shown in Table 1. Software can use these operations to provide the illusion of atomic execution - the execution of a region of code completely or not at all. The commit operation is used to denote both the beginning and the end of an atomic execution region (atomic region). It is used at the beginning of an atomic region to take a register checkpoint and to treat all future register and memory updates as speculative. It is used at the end of an atomic region to commit all speculative updates and discard the last checkpoint. The rollback operation is used to unconditionally abort an atomic region by restoring the last checkpoint. A rollback does not affect the program counter, so an instruction following the rollback can be used to redirect control flow as necessary.

Between checkpoints, all updates are speculatively written to either working registers or the data cache. If a cache line is speculatively written, its speculative bit is set and it is transitioned to the dirty state (after first evicting any non-speculative dirty data on the line into a victim cache). The hardware can commit speculative work in a single cycle by copying the working registers onto their shadowed counterparts and flash clearing all speculative bits in the data cache. Alternatively, the hardware can roll back all speculative work by restoring the working registers from their shadowed counterparts and flash. Our acceleration framework leverages the hardware support for speculative execution in Efficeon toHARDWARE SUPPORT FOR SPECULATION IN

## B.2 Bytecode Execution Time Breakdown

Table 1 shows the execution time breakdown for two benchmarks:DaCapo [BGH+06] and SPEC CPU2000 [Hen00]. The DaCapo benchmark is a popular set of open source, client-side Java benchmarks.

| **Benchmark** | **Interpreter** | **Translator** | **Non-tranlated** | **Translated** |
|---|---|---|---|---|
| **DaCapo** | 2% | 3.5% | 20.5% | 74% |
| **SPEC CPU2000** | 1% | 1% | 13% | 85% |

Table 1: Execution Time Breakdown for DaCapo and Spec 2000

Here is a brief description of each component of Table 1:

- **Interpreter:** The interpreter is used to reduce translation overhead of infrequently executed code, provide profiling information, interrupt narrowing, and serves as a low latency "forward progress engine". The CMS interpreter uses extensive hardware support, and is only 30x slower than native code execution.

- **Translator:** The translator is responsible for generating Efficeon code that corresponds to input x86 code.
- **CMS Non-translated Execution:** This component comprises the memory allocation, translation management, threading, TLB handler, translation lookup, and power management.
- **Translated Code:** The translated code execution includes execution of Efficeon code generated by the translator natively, including any CMS management tasks included in translations as native code such as branch chaining.

## B.3 Transmeta Code Morphing Software

Transmeta's proprietary Code Morphing Software (CMS) [DGB$^+$03] [Kla00b] is a product-quality and highly-tuned software layer that runs at the heart of the Efficeon processor that dynamically interprets, optimizes and translates x86 instructions (guest ISA) into internal VLIW instructions (host ISA). CMS sits below the OS and is totally transparent to the entire software stack. CMS perform analyzes of the stream of independent instructions, sniffs out interdependencies, and then reschedules and combines them into molecules that are then executable by the VLIW CPU.

CMS is designed as a staged dynamic compiler, like many other dynamic binary translation (DBT) systems. Initially, the interpreter decodes and executes x86 instructions sequentially, with careful attention to memory access ordering and precise reproduction of faults, while collecting data on execution frequency, branch directions, and memory-mapped I/O operations, as shown in Figure 4 on the next page. The interpreter is designed to provide a low-latency cold start and collect enough profiling information to build complex code regions in later stages.

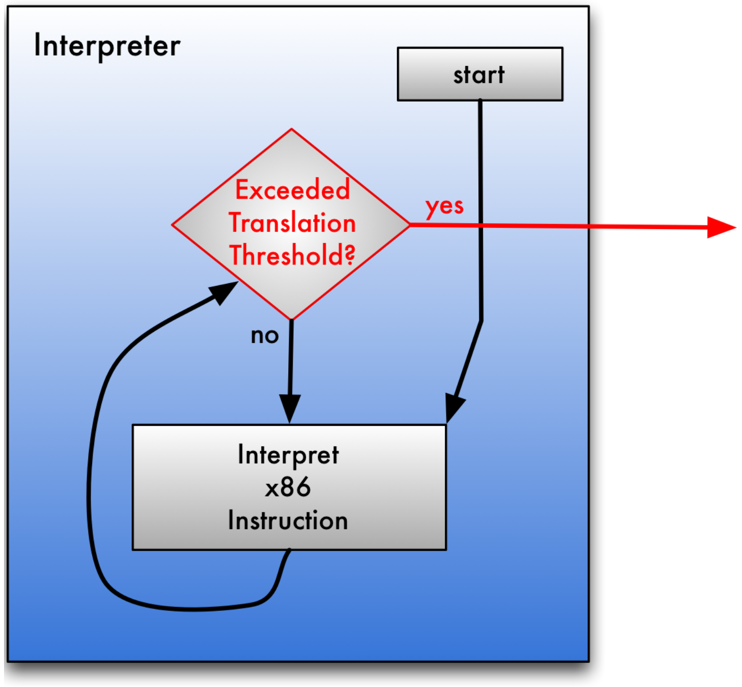


Figure 4: Interpretation and profiling of x86 Instructions

When the number of executions of a section of x86 code reaches a certain threshold, the dynamic compiler of CMS, which is called the translator, as shown in Figure 5 on the following page, promotes the hot code region to a second compilation stage (second gear) on which a lightweight-optimized code region is built for the host code starting at that particular address. The translator includes three levels of optimization aggressiveness (gears) that allow the translator to generate code quickly in order to minimize the latency of executing native code while applying aggressive optimizations to get higher performance code for code that is executed frequently. The optimizer implements both classical non-speculative optimizations (such as common sub-expression elimination) and speculative optimizations that exploit the rollback hardware, as illustrated in Figure 7 on page 147. If any speculative assumption fails, the hardware recovery mechanism will roll back the execution to the beginning of the atomic region that failed, and the execution is switched to the interpreter, to re-execute the code without speculation. This avoids the need to generate compensation recovery code. The translator also includes a register allocator and scheduler. Our

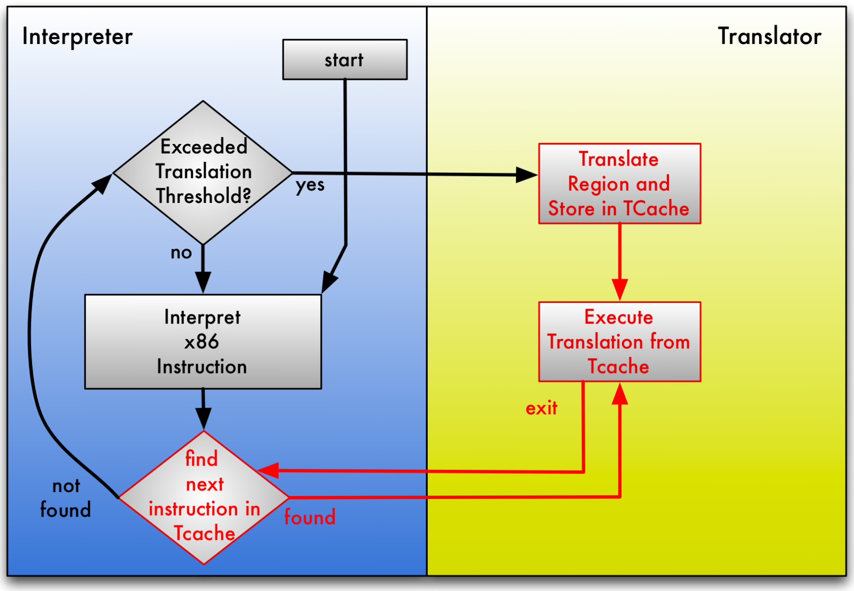

Interpreter
Translator
start
Exceeded Translation Threshold?
yes
no
Interpret x86 Instruction
Translate Region and Store in TCache
Execute Translation from Tcache
exit
find next instruction in Tcache
not found
found

Figure 5: Translation of x86 Instructions

bytecode accelerator leverages the rollback hardware in handling exceptions and interruptions. This simplifies our accelerator since in the event of a trap, the handling is always performed in the architectural ISA at OS level. This allows the implementation of a very light-weight bytecode runtime module inside the DBT.

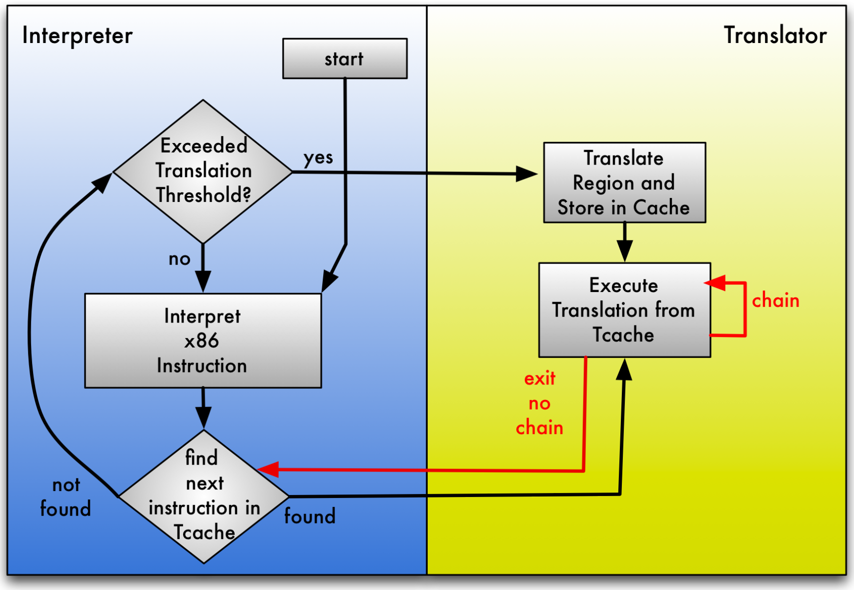

Interpreter
Translator
start
Exceeded Translation Threshold?
yes
no
Interpret x86 Instruction
Translate Region and Store in Cache
Execute Translation from Tcache
chain
exit no chain
find next instruction in Tcache
not found
found

Figure 6: Chaining X86 Translations

Once a translation has finished execution, it will exit by jumping somewhere else in the DBT runtime, which will then find the next translation entry point to execute. If a translation point is found, the translation cache will chain the code in the translation that jumped to it. The chaining process, shown in Figure 6 on the preceding page, alters the translation's code so that it will subsequently jump directly to the next entry point to execute. If no translation is found, the execution jumps back to the interpreter. Our acceleration scheme also leverages the chaining mechanism of Efficeon, allowing two execution traces to be chained according to the flow of execution.

This unique combination of hardware and software allows the processor to be more efficient, adding intelligence to the Efficeon processor in order to manage power consumption and heat dissipation in ways not found in other x86 microprocessors.

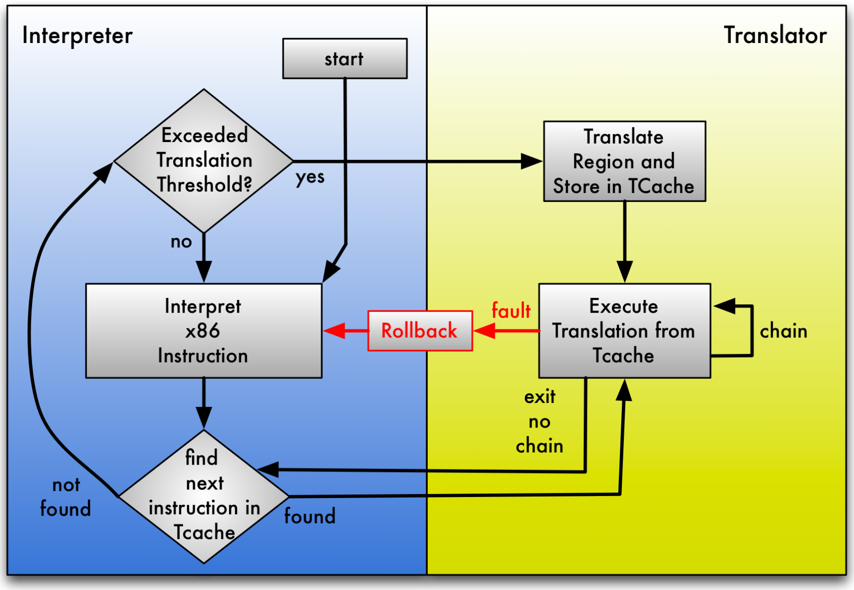

Interpreter
Translator
start
Exceeded Translation Threshold?
yes
no
Translate Region and Store in TCache
Interpret x86 Instruction
Rollback
fault
Execute Translation from Tcache
chain
exit no chain
not found
find next instruction in Tcache
found

Figure 7: Speculative Support for Agressive Optimizations

Hardware/Software co-designed processors, like the Transmeta Efficeon, in general eschew branch prediction and scheduling hardware that is common in most mainstream processors and instead rely on software to fulfill this task. Hence, the core of VLIW processors has more die space for further functional units, registers or cache memory units, as opposed to an equivalent CPU die using the mainstream architectures.

Using software to decompose complex instructions into simple atoms, and to schedule and optimize the atoms for parallel execution saves millions of logic transistors and cuts power consumption on the order of 60-70% over conventional approaches - while at the same time enabling aggressive code optimization techniques that are simply not feasible in traditional x86 implementations [Kla00b]. Transmeta's CMS and fast VLIW hardware, working together, achieve low power consumption without sacrificing high performance for real-world applications. This was the main motivation for us to use this processor in our experimentations.

## C Translation Example

In this appendice we show an example of translation for the function execute() of Loop benchmark, from the CaffeineMark 3.0 set of benchmarks.

```
public int execute()
{
      this.fibs[0] = 1;
      for (int i = 1; i < this.FIBCOUNT; i++) {
          this.fibs[i] = (this.fibs[(i - 1)] + i);
      }
      int i1 = 0;
      int i2 = 0;
      for (int j = 0; j < this.FIBCOUNT; j++) {
          for (int k = 1; k < this.FIBCOUNT; k++)
          {
              int m = this.FIBCOUNT + this.dummy;
              i1 += m;
              i2 += 2;
              if (this.fibs[(k - 1)] < this.fibs[k])
              {
                  int n = this.fibs[(k - 1)];
                  this.fibs[(k - 1)] = this.fibs[k];
                  this.fibs[k] = n;
              }
          }
       }
       this.sum1 = i1;
       this.sum2 = i2;
       return this.fibs[0];
}
```

Listing 5: Execute function from Loop Benchmark in CaffeineMark 3.0

```
001a0e: 52a4 1900  |0031: iget v4, v10, LLoopAtom;.FIBCOUNT:I  // field@19
001a12: 52a5 1a00  |0033: iget v5, v10, LLoopAtom;.dummy:I   // field@1a
001a16: b054       |0035: add-int/2addr v4, v5
001a18: b041       |0036: add-int/2addr v1, v4
001a1a: d800 0002  |0037: add-int/lit8 v0, v0, #int 2 // #02
001a1e: 54a4 1b00  |0039: iget-object v4, v10, LLoopAtom;.fibs:[I //field@1b
001a22: 9105 0308  |003b: sub-int v5, v3, v8
001a26: 4404 0405  |003d: aget v4, v4, v5
001a2a: 54a5 1b00  |003f: iget-object v5, v10, LLoopAtom;.fibs:[I //field@1b
001a2e: 4405 0503  |0041: aget v5, v5, v3
001a32: 3554 1600  |0043: if-ge v4, v5, 0059 // +0016
```

Listing 6: Bytecode trace for lines 12-15 of Listing 5

```
[ begin b0 ]
  0xfed2e790: { pre au,xunlock,unlock,here,cmit,431 }
  0xfed2e794: { ld.32/1 r6,x86:[r7+40] ; ld.32/0 r42,x86:[r7] }
  0xfed2e79c: { ld.32/1 r40,x86:[r7+4] }
  0xfed2e7a0: { add.32/0 r43,r42,2; brc ~p3, 0xffd6a9f0 <mtc_chain_0>;
                ld.32/0 r36,x86:[r7+32]; tst.c.32/1 p3=ne,r6,r6 }


[ begin b2 ]
  0xfed2e7b0: { ld.32/0 r37,x86:[r6+12] ; ld.32/1 r38,x86:[r6+16] }
  0xfed2e7b8: { ld.32/0 r3,x86:[r7+12] ;
                ld.32/1 r1,x86:[r6+8] ; alias 0,0,0,0,15 }
  0xfed2e7c4: { add.32/1 r39,r38,r37 ; alias 0,1,0,15,15 ;
                st.32/1 r43,x86:[r7] }
  0xfed2e7d0: { sub.32/0 r0,r3,r36 ; add.32/1 r41,r40,r39 }
  0xfed2e7d8: { tst.c.32/0 p3=ne,r1,r1 ; sh2add.32/1 r34,r0,r1 ;
                segaddx/0 r31,x86:[r62+r1] }
  0xfed2e7e4: { brc ~p3, 0xffd6ab30 <mtc_chain_1> ;
                alias 0,1,0,15,15 ; st.32/1 r41,x86:[r7+4] }
```

```
[ begin b4 ]
  0xfed2e7f0: { ld.32/0 r35,x86:[r1+8] }
  0xfed2e7f4: { ld.32/1 r19,x86:[r34+12] }
  0xfed2e7f8: { cmp.c.32/0 p3=ltu,r0,r35 ;
                brc ~p3, 0xffd6ab58 <mtc_chain_2> }

[ begin b6 ]
  0xfed2e800: { or.32/0 r1,r62,r19; or.32/1 r0,r19,0 }
  0xfed2e808: { tst.c.32/0 p3=ne,r31,r31; brc ~p3, 0xffd6ab50 <mtc_chain_3>;
                [dup tst.c.32/0 p3=ne,r31,r31]; or.32/1 r6,r62,r31 }

[ begin b8 ]
  0xfed2e818: { cmp.c.32/0 p3=ltu,r3,r35; brc.h p3, 0xffd6ab40 <mtc_chain_4> }

[ begin b9 ]
  0xfed2e820: { execf1 nop; br 0xffd6ab28 <mtc_chain_5> }
```

Listing 7: JIT/CMS translation for bytecode trace of Listing 6

```
[ begin b0 ]
  0xfecd8bc8:  { pre aa,unlock,here,cmit,392 }
  0xfecd8bcc:  { ld.32/1 r46,x86:[r7] ; ld.32/0 r19,x86:[r7+40] }
  0xfecd8bd4:  { ld.32/1 r20,x86:[r7+12] ; ld.32/0 r40,x86:[r7+32] }
  0xfecd8bdc:  { or.32/1 r45,r62,r46 ; add.32/0 r46,r46,2 }
  0xfecd8be4:  { or.32/1 r44,r62,r46 ; cate.32/0 53,r19,0 ;
                 ld.32/1 r43,x86:[r19+8] ; alias 0,0,0,0,15 ;
                 [dup ld.32/1 r43,x86:[r19+8]];
                 segaddx/0 r42,x86:[r62+r19] }
  0xfecd8bfc:  { or.32/1 r39,r62,r20 ; or.32/0 r37,r62,r40 ;
                 st.32/1 r46,x86:[r7] ; alias 0,1,0,1,15 }
  0xfecd8c0c:  { or.32/1 r46,r19,0 ; or.32/0 r41,r62,r43 }
  0xfecd8c14:  { or.32/1 r46,r20,0 ; alias 0,2,0,2,15 ;
                 ld.32/1 r36,x86:[r43+8] ; cate.32/0 53,r43,0 }
  0xfecd8c24:  { sub.32/1 r46,r20,r40 ; alias 0,0,0,3,15 ;
                 ld.32/1 r18,x86:[r42+8] }
  0xfecd8c30:  { or.32/0 r38,r62,r46 ; sh2add.32/1 r35,r46,r41 ;
                 alias 0,0,0,4,5 ; ld.32/1 r40,x86:[r42+16] ;
                 ld.32/0 r20,x86:[r42+12] }
  0xfecd8c44:  { or.32/1 r33,r62,r35 ; add.32/0 r46,r39,1 }
  0xfecd8c4c:  { sh2add.32/1 r35,r39,r41 ; alias 0,2,0,6,15 ;
                 ld.32/1 r34,x86:[r35+12] ; or.32/0 r28,r62,r46 ;
                 [dup ld.32/1 r34,x86:[r35+12]] ;
                 segaddx/0 r23,x86:[r62+r18] }
  0xfecd8c64:  { catnb.32/1 53,r38,r36 ; add.32/0 r46,r20,r40 }
  0xfecd8c6c:  { or.32/1 r31,r62,r34 ; alias 0,2,0,7,15 ;
                 ld.32/1 r19,x86:[r35+12] ; or.32/0 r26,r62,r46 ;
                 [dup ld.32/1 r19,x86:[r35+12]] ;
                 segaddx/0 r30,x86:[r62+r35] }
  0xfecd8c84:  { ld.32/1 r46,x86:[r7+4] ; alias 0,0,0,1,15 ;
                 catnb.32/1 53,r39,r36 }
  0xfecd8c90:  { cmp.32/0 p3=ge,r31,r19 ; or.32/1 r34,r19,0 ;
                 alias 3,61,0,15,15 ; st.32/1 r31,x86:[r7+16] }
  0xfecd8ca0:  { alias 3,0,61,15,15 ; brc p3, 0xffd6a9f0 <mtc_chain_0> ;
                 add.32/0 r46,r46,r26 ; or.32/1 r29,r62,r19 ;
                 st.32/0 r19,x86:[r7+20] }
```

Listing 8: AccelDroid translation for bytecode trace of Listing 6