CO-OPTIMIZING HARDWARE DESIGN AND META-TRACING JUST-IN-TIME COMPILATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Berkin Ilbeyi

May 2019

© 2019 Berkin Ilbeyi

CO-OPTIMIZING HARDWARE DESIGN AND
META-TRACING JUST-IN-TIME COMPILATION

Berkin Ilbeyi, Ph.D.
Cornell University 2019

Performance of computers has enjoyed consistent gains due to the availability of faster and cheaper transistors, more complex hardware designs, and better hardware design tools. Increasing computing performance has also led to the ability to develop more complex software, rising popularity of higher level languages, and better software design tools. Unfortunately, technology scaling is slowing, and hardware and software depend more on each other to continue deliver performance gains in the absence of technology scaling. As single-threaded computing performance slows down, emerging domains such as machine learning are increasingly using custom hardware. The proliferation of domain-specific hardware requires the need for more agile hardware design methodologies. Another trend in the software industry is the rising popularity of dynamic languages. These languages can be slow, but improvements in single-threaded performance and just-in-time (JIT) compilation have improved the performance over the years. As single-threaded performance slows down and software-only JIT techniques provide limited benefits into the future, new approaches are needed to improve the performance of dynamic languages.

This thesis aims to address these two related challenges by co-optimizing hardware design and meta-tracing JIT compilation technology. The first thrust of this thesis is to demonstrate meta-tracing JIT virtual machines (VMs) can be instrumental in building agile hardware simulators across different abstraction levels including functional level, cycle level, and register-transfer level (RTL). I first introduce an instruction-set simulator that makes use of meta-tracing JIT compilation to productively define instruction semantics and encodings while rivaling the performance of purpose-built simulators. I then build on this simulator and add JIT-assisted cycle-level modeling and embedding within an existing simulator to achieve a very fast cycle-level simulation methodology. I also demonstrate that a number of simulation-aware JIT and JIT-aware simulation techniques can help productive hardware generation and simulation frameworks to close the performance gap with commercial RTL simulators. The second thrust of this thesis explores hardware acceleration opportunities in meta-tracing JIT VMs and proposes a software/hardware co-optimization scheme that can significantly reduce dynamic instruction count in meta-tracing JIT VMs. For this, I first present a methodology to study and research meta-tracing JIT VMs and perform a detailed cross-layer workload characterization of these VMs. Then I quantify a type of value locality in JIT-compiled code called object dereference locality, and propose a software/hardware co-optimization technique to improve the performance of meta-tracing JIT VMs.

BIOGRAPHICAL SKETCH

Berkin İlbeyi was born on September 26, 1989 in Istanbul, Turkey. He is the only child of Uğur and Gül İlbeyi. He grew up in Istanbul. He attended FMV Ayazağa Işık Lisesi from kindergarten until the 8th grade and Robert College for high school. During high school, he first discovered programming by writing games on his TI-84 Plus calculator. He took Java programming and electronics elective classes and was fascinated by the limitless possibilities of software and hardware design at an early age. At this time, he also took lessons for oil painting and violin and enjoyed these activities. He attended Lafayette College in Easton, PA, double majoring in Electrical and Computer Engineering, and Computer Science. His passion for software and hardware grew as he took advanced classes. His summer internship at EPFL in Lausanne, Switzerland, working with Prof. Babak Falsafi introduced him to computer architecture research, and he decided to do a PhD in computer architecture.

Berkin was accepted to Cornell University in 2012. At Cornell, he was advised by Prof. Christopher Batten. He initially worked on the explicit loop specialization project with Shreesha Srinath. Later, he helped write a funded grant on hardware acceleration techniques for Python, which funded the majority of his PhD. He worked on a number of projects at the intersection of meta-tracing just-in-time compilers and hardware design that are detailed in this dissertation. During his PhD, he had two internships: one at Google Platforms working on a prototype FPGA backend for the XLA compiler, and another at Oracle Labs working on the Truffle/Graal system. He also frequently played, and still very much enjoys, squash. Berkin has matured as a researcher, engineer, and as a person during this time, and he is very thankful to everybody who has supported him during this challenging but rewarding adventure. Berkin will start working at Google upon the completion of his PhD.

This document is dedicated to my parents and Jocelyn.

ACKNOWLEDGEMENTS

This thesis became a reality with the support of numerous people in my life. First and foremost, my advisor, Christopher Batten, was tremendously influential not only for my research direction, but also for the development of my professional skills. His vision on vertically integrated computer architecture research was what attracted me to Cornell and it has been a fascinating ride. He remains one of my biggest role models with his technical knowledge, intellect, ability to distill and teach complicated concepts, writing and presenting style, and skill in asking the right (often hard) questions. I am excited to find out what Chris and the Batten Research Group achieve in the future. I hope that our paths coincide and get to work together again!

I would like to thank Carl Friedrich Bolz-Tereick. He has a wealth of knowledge for everything PyPy, dynamic languages, and JITs in general. He has been a great collaborator for many of the works presented in this thesis so his help was invaluable. There were also many other teachers and professors who helped and guided me to reach this point. My committee members José Martínez and Zhiru Zhang helped me refine my research direction with their thoughtful feedback. I also enjoyed the classes I took from them. Additionally, Compilers and Programming Languages classes I took from Andrew Myers and Adrian Sampson respectively were excellent. Also many thanks to my professors in college, especially John Nestor, John Greco, and Chun Wai Liew. The classes they taught and being mentored by them stirred my intellectual curiosity, leading me to pursue a PhD degree. Furthermore, many thanks to my high-school teachers, Alison Oğuz, Joe Welch, Cengiz Ağalar, Sandra Çorbacıoğlu, and many others for inspiring to be creative and introducing me to the fascinating world of engineering, programming, and electronics.

I would like to acknowledge the members of the Batten Research Group (BRG). In particular, I collaborated with four BRG members very closely and each of these experiences have been very enjoyable. Firstly, many thanks to Shreesha Srinath for being a great friend and collaborator. I really enjoyed working on the XLOOPS project together. Whiteboarding sessions, attending conferences together, and grabbing beers after work to rant about research and life were some of the highlights of graduate school. Secondly, I collaborated with Derek Lockhart on the Pydgin project and it was a lot of fun to hack PyPy together to support more ISAs and get great performance. He also mentored me during my internship at Google and remains a good friend. I collaborated with Shunning Jiang on the Mamba project, and working together to make PyMTL go fast was a great experience. Lastly, Lin Cheng helped me with the skippable sub-traces project, and I enjoyed working with him. I am also excited to see where he takes the project as he takes ownership over co-optimization techniques to make PyPy go fast. I thank the other members of BRG as well: Ji Kim led the Loop Task Accelerator project that I was briefly a part of (and I had a lot of fun playing board games with him); Chrisopher Torng led the BRG Test Chip 2 (and I always enjoyed chatting with him about research and career); and despite I did not collaborate as much with the rest of BRG, Moyang Wang, Khalid Al-Hawaj, Tuan Ta, Yanghui Ou, and Peitian Pan, I appreciate their friendship and support. I would also like to acknowledge the members of Cornell Computer Systems Laboratory, Skand Hurkat, Andrew Ferraiuolo, and many others, for their support. Additional thanks to Yunsup Lee, Sarah Mount, Maciej Fijalkowski, Alex Katz, Mohammad Dohadwala, Scott McKenzie, and Alvin Wijaya for other technical help and feedback on the work presented in this thesis. A more detailed discussion of the technical contributions by my collaborators can be found in Section 1.5. This work was supported in part by NSF CRI Award #1512937, NSF SHF Award #1527065, NSF XPS Award #1337240, NSF CAREER Award #1149464, DARPA POSH Award #FA8650-18-2-7852, DARPA Young Faculty Award #N66001-12-1-4239, AFOSR YIP Award #FA9550-15-1-0194, and equipment, tool, and/or physical IP donations from Intel, Xilinx, Synopsys, Cadence, and ARM.

This degree would definitely not be possible without my parents and their constant love, affection, and support. Their financial support enabled me to attain a great elementary school education, attend the best high school in Turkey, and then make my dream of studying in the US come true. Supporting my education meant that they had to make significant sacrifices to their quality of life they could have otherwise had. Besides financial support, their emotional support in difficult times also was crucial. I will be forever thankful to them.

Lastly, many thanks to Jocelyn for the unlimited support she has provided. Even though I have met her only in the last two years of graduate school, she became an unreplaceable part of my life. I am looking forward to a life together with her.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

JIT	just-in-time (compilation)
VM	virtual machine
IR	intermediate representation
AOT	ahead-of-time (compilation)
FL	functional level
CL	cycle level
RTL	register-transfer level
PC	program counter
CAD	computer-aided design
FPU	floating point unit
ALU	arithmetic and logic unit
SOC	system-on-chip
GPU	graphics processing unit
HLS	high-level synthesis
API	application programming interface
HLR	high-level representation
DIR	directly interpretable representation
DER	directly executable representation
OSR	on-stack replacement
AST	abstract syntax tree
RF	RPython framework
RL	RPython language
RR	RPython runtime
RTT	RPython translation toolchain
ISA	instruction set architecture
DSL	domain-specific language
ISS	instruction-set simulator
IPS	instructions per second
MIPS	millions of instructions per second
DBT	dynamic binary translation
ADL	architecture description language
FF	fast forwarding
NNFF	native-on-native fast forwarding
JIT-FFE	JIT-assisted fast-forward embedding
JIT-FFI	JIT-assisted fast-forward instrumentation
BBV	basic block vector
LRU	least recently used
HDL	hardware description language
HGF	hardware generation framework
HGSF	hardware generation and simulation framework
DAG	directed acyclic graph
TLB	transaction lookaside buffer

CLBG	computer languages benchmark suite
GIL	global interpreter lock
PAPI	performance application programming interface
GC	garbage collector/collection
IPC	instructions per cycle
MPKI	misses per thousand instructions
SST	skippable sub-traces
ODL	object dereference locality
DVL	dependent value locality
IVL	IR-level value locality
DFG	data flow graph
HP	helper processor
SSIE	SST synchronous invalidation engine
ET	entry table
STEBA	sub-trace entry base address register
STEA	sub-trace entry address
STEPC	sub-trace end PC
NR	no-recompile
RO	recompile-only
BRGTC2	Batten Research Group test chip 2

CHAPTER 1
INTRODUCTION

The performance of a computer program depends on many factors: algorithms, programming languages, compilers, and the underlying hardware. The performance of the underlying hardware in turn depends on the process technology, architectures, and the computer-aided design (CAD) tools. All of these factors have co-evolved significantly since the era of mainframes and minicomputers into today's wide spectrum of computer systems from embedded to cloud computing. This co-evolution has been primarily fueled by Moore's Law and Dennard Scaling, which have delivered exponentially faster, smaller, and cheaper transistors. However, with Dennard Scaling breaking down, and Moore's Law slowing down, better hardware design tools and software/hardware co-design are increasingly important to ensure that computer systems meet demands of the future.

1.1 Software/Hardware Co-Evolution

Figure 1.1 shows the software and hardware co-evolution trend over the past several decades. As predicted by Moore's Law, new process technologies reduced transistor feature sizes by a constant ratio every 18 months to two years for roughly the same dollar budget [Moo65]. This effectively leads to exponentially more transistors in a chip for the same cost. Similarly, Dennard Scaling outlined a constant-factor scaling down of feature sizes, voltages, and gate capacitances [DGY$^+$74], which leads to exponentially faster transistors for roughly the same power budget. These two factors combined gave hardware designers more and better transistors.

More transistors meant that more features could be moved to the same chip, increasing the level of integration. Transitioning from mini-computers to micro-computers, more of the hardware could be concentrated to fewer chips, until the entire processor could be built in a single chip. In addition to building hardware for general-purpose computation, some of the transistors have been specialized to enable hardware support for emerging domains. Initially, domain-specific units such as the floating point unit (FPU) could be moved on-chip. More recently, systems-on-chip (SOCs) combined all of the computing resources that an embedded system needs into a single chip. More transistors also enabled more aggressive designs to improve the performance. Multiple arithmetic and logic units (ALUs) were added to processors, additional auxiliary memories and logic were added to coordinate multiple instructions to be executed concurrently, out-of-order, and in deep

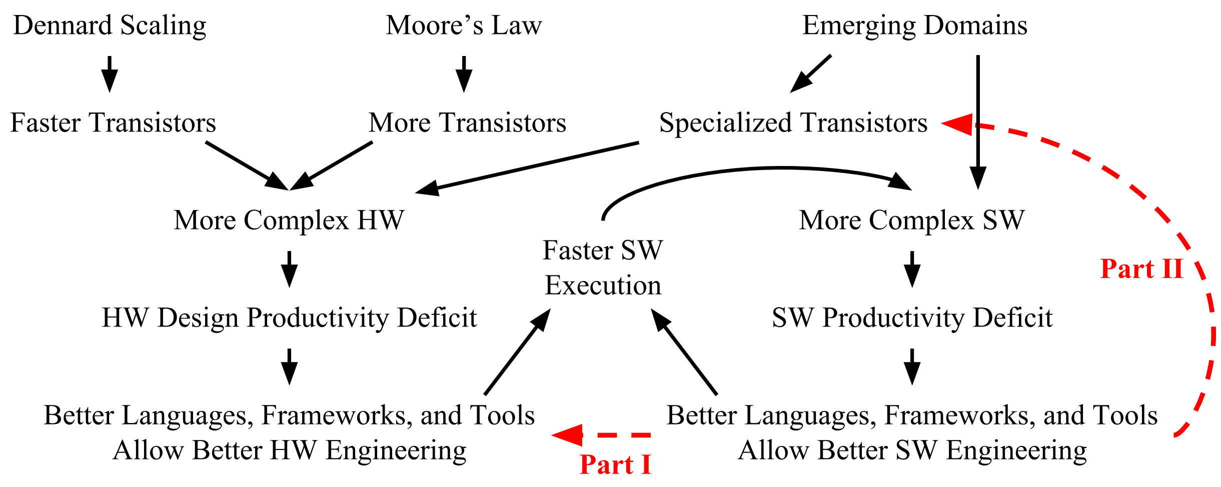


Figure 1.1: Software/Hardware Co-Evolution – The Part I arrow represents using state-of-the-art (JIT) compilers to allow building better hardware design tools and is the first part of this thesis; the Part II arrow represents building specialized hardware accelerators for higher level (dynamic) programming languages and is the second part of this thesis.

pipelines [SS95, TEL95]. Sophisticated branch and value prediction techniques were implemented, bookkeeping hardware was added to allow rollbacks in case of misspeculations [Smi81]. The addition of many levels of coherent caches enabled memory systems to navigate the latency and capacity trade-off. Homogeneous and heterogeneous many-core systems and graphical processing units (GPUs), with scalable networks-on-chip, led to highly parallel hardware for workloads that can take advantage of parallelism.

While hardware itself was getting cheaper, the design of the hardware was getting more expensive. The additional hardware to design and verify required the use of more numerous but novice hardware designers. Whereas hardware design at its infancy only required small teams of expert designers, the new hardware design paradigm required coordinating hundreds of engineers at various skill levels working on various aspects of design space exploration, implementation, and verification. Unfortunately, coordinating large teams of hardware designers is difficult, leading to a *productivity deficit*. Coordinating these designers requires raising the level of abstraction of hardware design. Instead of drawing polygons, first gate-level design, then register-transfer-level (RTL) design became more popular. More recently, even higher levels of design such as high-level synthesis (HLS) are gaining traction [CLN+11].

One downside of hardware design using higher levels of abstractions is that it can lead to a *performance deficit*, where the hardware can be inferior in performance, area, or energy than one designed in a lower level of abstraction. This is especially true when a higher level of abstraction is first developed, and the tools are at their infancy. However, high-level design is certainly much more scalable across large teams of varied skill levels and enable better hardware engineering. Over time, better computer-aided design (CAD) tools and algorithms are developed that target the higher levels of abstraction, and close a major part of the performance deficit. With the help of advanced CAD tools, hardware designed at a higher level is very competitive with, and often better than, hardware designed at a lower level. To this day, some extremely performance-critical parts of the chip have to be designed at gate- or polygon-level, but an average hardware designer seldom works at these levels. With the combination of better hardware engineering, better CAD tools afforded through better software engineering, and more transistors, increasingly faster computers were built.

A similar trend exists in software design as well. As computers were getting faster for the same budget, legacy programs could be executed faster. The availability of cheap computing resources led to the ability and demand to automate more tasks from different industries that were previously done manually. More businesses started relying on electronic bookkeeping, reporting, and analysis of their data. The proliferation of software created its own demand for more systems software development to enable easier development of software and for programs to better work with each other. Virtual memory support had to be implemented to coordinate multiple programs to run reliably at the same time. Multi-user, time-sharing operating systems had to be developed so that multiple users can use the same machine as if they are the only ones logged in. Networking and the internet required efficient sharing of data across a diverse set of hardware. The rise of bad actors required re-thinking the software architecture with a heavier emphasis on security and heavier use of encryption. Graphical user interfaces and computer graphics required a new set of software technologies to be developed (e.g., Self [HU94b]).

This led to a large proliferation of software engineers to meet the software development demand. A higher proportion of these developers were novice developers and the tools and languages of the time, e.g. assembly programming, were not scalable across large teams. This created a productivity deficit in the software world as well. To bridge this deficit, higher level languages were developed. Languages such as C required less boilerplate code to be written while still staying close to the machine. These languages also enabled limited portability across multiple hardware architectures. Later, object-oriented programming, e.g., with C++, enabled representing program logic and data at a higher level. There was also a trend towards better defined application programming interfaces (APIs), data encapsulation, permission management, inheritance techniques to reduce code duplication, and version control systems to enable more effective collaboration. Later still, automatic memory management, e.g., with Java, made programming safer and more productive. More recently, dynamic programming languages have been rising in popularity mostly due to their advanced productivity features.

Analogous to the raised abstraction levels in hardware design, using higher level languages causes a performance deficit. Especially during the infancy of a new language, there could be significant performance degradation compared to a low-level language. For example, when C was first developed, it did not have an optimizing compiler, so the performance could be much worse than writing assembly. Similarly, Java originally did not have a just-in-time (JIT) compiler, so its performance was much worse than C++. However, high-level languages enabled scalable productivity for large teams of software developers, leading to better software engineering. This, in turn, enabled the development of compilers with advanced optimizers. Better optimizing compilers allow faster execution of programs, and importantly they close a major part of the performance deficit. Today, assembly-level programming, just like drawing polygons, is often used only in the most extremely performance critical parts of modern software, while the vast majority of programmers write code at much higher levels.

It is important to note the synergy between hardware and software in this co-evolution. More and more problems encountered by humans can be automated and solved using software running on ever-faster processors. The software engineering improvements and higher level languages lead to the development of more sophisticated CAD algorithms. These CAD algorithms can close the performance deficit by raising the abstraction level used for hardware design. Designing hardware in higher abstraction levels often outperforms hardware designed using lower abstraction levels. Newer hardware design methodologies can help pick more optimal choices in the large design space, and larger chips to be designed and verified at scale. With better hardware engineering and improvements in process technology allows building faster computers, and this co-evolution continues as long as Moore's Law continues to deliver more transistors.

1.2 The Present and Future of Hardware Design

Dennard Scaling has long stopped delivering faster transistors since the early 2000s. Furthermore, Moore's Law has been slowing down. Below, I present some of my observations and predictions on how this might affect hardware design.

Large hardware designs will continue to be economical. Even though Moore's Law is slowing down, there is likely plenty of unrealized potential with the current economics of chips. Over the decades, the pace of hardware design was not able to catch up with the available transistors for a given cost. With better hardware engineering and domain-specific specialization, the currently available transistors could be used more effectively. Furthermore, with newer process technologies becoming far in between, slightly older technologies, which are still very competitive, might get commoditized. As the research and development costs for these technologies are recouped, the price per transistors will likely continue to fall into the future.

Single-threaded performance gains will come to an end. Since the end of Dennard Scaling and stagnating maximum clock frequencies, the performance of single-threaded applications continued to improve. This was achieved using larger hardware designs with the addition of larger caches, wider networks, more accurate but larger branch predictors, larger issue windows, etc. Furthermore, with the help of better hardware engineering, the existing designs could be refined for more effective uses of the transistors. Unfortunately, the number of tricks available to hardware engineers is running out, and single-threaded performance improvements will eventually come to an end.

Domain specific designs will continue to gain importance. With the slowing single-threaded performance gains, yet relatively cheap transistors, more and more domains will invest in domain-specific hardware. Recently, machine learning had a significant amount of success building custom hardware [CDS+14, JYP+17]. Building custom hardware could become the most economic way of improving the performance and/or reducing the energy of different domains' computing needs.

Newer hardware design tools are emerging. Given that hardware engineering is becoming more widespread, the designs more varied, and performed by smaller teams, the productivity deficit continues to be a major issue. To address this, higher level abstractions and design tools are being introduced. For example, new hardware generation frameworks such as Chisel [BVR+12], Genesis2 [SAW+10], and PyMTL [LZB14] allow higher level software languages to be used for hardware design. Cycle-level simulators such as ZSim [SK13] and Sniper [CHE11] make use of dynamic binary translation technology to enable very fast simulations as long as the host and guest instruction set architectures (ISAs) match. Furthermore, approaches such as high-level synthesis (HLS) automatically generate hardware from untimed algorithms [CLN$^+$11]. The trend for newer methodologies and tools will likely continue to gain importance.

Challenge: More agile hardware design methodologies are needed. While newer hardware design methodologies have promise in increasing productivity to meet the demands of increased and more diverse hardware engineering challenges, these approaches still have a number of practical shortcomings. To name a few, the hardware generation frameworks often do not support fast simulations in the high-level language but require simulating in lower level Verilog, fastest functional and cycle-level simulation methodologies do not allow simulating guest ISAs that are different than the host ISAs, and current HLS approaches still typically require heavy designer involvement in generating the desired hardware. These challenges currently prohibit the new hardware design tools to be truly *agile*. Agile hardware design methodology and tools combine flexibility, productivity, and performance. Agile hardware design, just like agile software design, can be crucial for hardware engineering teems to quickly explore and iterate many different design choices at different levels of hardware design, and eventually refine their designs to hardware. But to achieve this, the practical issues in newer hardware design tools need to be solved.

1.3 The Present and Future of Software Performance

The software design and performance will also continue to evolve into the future. The slowing of Moore's Law will also have repercussions on software performance.

Software will continue to get larger and more complex. Humans will continue to have new and more complex problems, and writing software will continue to be the primary way to automate solutions to those problems. Computing devices are currently the most ubiquitous and inexpensive that they have ever been, and the availability of cheap and fast hardware creates an increasing demand to write more, larger, and more complicated programs. The increasing diversity and complexity of software itself is a challenge that is addressed with larger and more complex systems software. I do not expect the pace of software development to slow down soon.

Higher level languages will continue to be popular. As the demand for software grows, and more people than ever develop software, higher level languages will continue their rise in popularity. This is already being experienced with dynamic languages such as Python and JavaScript being used more widely and for a wider number of domains. Dynamic languages use dynamic type systems that can allow rapid prototyping, but also typically have other productivity-oriented features such as: lightweight syntax, managed memory, rich standard libraries, support for dynamic code execution, interactive execution environments, and advanced reflection capabilities.

Just-in-time (JIT) compilation will be more important. Higher level languages are usually slower than lower level languages. This is especially true for dynamic languages. As more code is being developed using higher level languages, and since single-threaded performance stops delivering performance improvements, the performance of these languages will be an issue. Luckily, JIT compilation has seen a lot of success and wider adoption over the last decade. One drawback of JIT compilation is the engineering challenge to develop, maintain, and extend state-of-the-art JIT compilation frameworks. However, there have been successful projects that use partial evaluation and meta-tracing to repurpose a JIT compiler across multiple languages (see Chapter 2 for more details). These approaches will become more important for achieving the productivity benefit of higher level language while approaching the performance of lower level languages.

JIT compilation will have diminishing performance improvements. JIT compilation unlocks significant performance benefits compared to simpler ways to execute higher level languages, such as interpreters. A JIT can typically achieve $10\times$ or better performance compared to a fast interpreter (see Chapter 6). Unfortunately, software-only improvements in JIT technology will likely yield diminishing returns as low-hanging fruit opportunities are exploited.

Challenge: Novel JIT execution methods are needed. While higher level languages enable the development of diverse and ambitious software, JIT compilation and improving single-threaded performance have been the best tools in ensuring these languages maintained their performance appeal. With both of these factors slowing down, there is a challenge in ensuring high-level languages to be performance competitive with lower level languages. Hardware specialization for the execution of high-level and dynamic languages can be a unique opportunity to address this challenge.

1.4 Thesis Overview

This thesis presents approaches to address the agile hardware design and novel JIT execution challenges by co-optimizing meta-tracing JIT VMs and hardware design. This thesis has two thrusts that tackle two opposite but related problems: (1) accelerating agile hardware simulations with meta-tracing JIT VMs, and (2) building hardware accelerators for meta-tracing JIT VMs. The first thrust can be seen as using JIT compilation technology to allow building better hardware design tools, as seen in the new Part I arrow in Figure 1.1. The second thrust can be seen as building hardware specialization not for new application domains, but for generally targeting high-level, dynamic languages, as seen in the new Part II arrow in the figure.

Since meta-tracing is a key enabler for the contributions of this thesis, Chapter 2 presents an extensive background on dynamic languages, JIT compilation, partial evaluation, and meta tracing. The rest of the thesis is organized in two parts. Part I presents the first thrust and demonstrates how meta-tracing JIT VMs can be used to accelerate functional-level (FL), cycle-level (CL), and register-transfer level (RTL) hardware simulations. Chapter 3 presents Pydgin, where we have built an agile instruction-set simulator; Chapter 4 presents PydginFF, where I introduce fast-forward embedding and instrumentation to Pydgin to speed up sampling-based agile cycle-level simulation; and Chapter 5 presents the Mamba techniques, which are simulation-aware JIT and JIT-aware simulator techniques to speed up the new version of PyMTL, an agile hardware generation and simulation framework [LZB14]. Part II of the thesis is dedicated to the second thrust of building hardware accelerators for meta-tracing JIT VMs. Chapter 6 provides a comprehensive cross-layer workload characterization of meta-tracing JIT VMs to identify hardware acceleration opportunities, and Chapter 7 identifies object dereference locality and intermediate-representation-level value locality in JIT-compiled codes of meta-tracing JIT VMs and software/hardware co-design mechanisms to exploit these patterns. Finally, Chapter 8 concludes this thesis.

The primary contributions of this thesis are:

- I demonstrate meta-tracing JIT VMs can be used to speed up agile FL simulation of processors.
- I extend agile FL simulation using JIT fast-forward embedding and instrumentation for sampling-based agile CL simulation of computer systems.
- I present JIT-aware hardware generation and simulation framework (HGSF) and HGSF-aware JIT mechanisms for accelerating agile RTL hardware simulations using meta-tracing JIT VMs.

- I present a novel methodology to study meta-tracing JIT VMs and allow experimenting with hardware acceleration techniques.
- I perform an extensive cross-layer workload characterization of meta-tracing JIT VMs.
- I identify and quantize object dereference locality (ODL) and intermediate-representation-level value locality (IVL) in JIT-compiled code in the context of meta-tracing JIT VMs.
- I propose a software/hardware co-design approach to exploit ODL and IVL.

1.5 Collaboration, Previous Publications, and Funding

This thesis was made possible by the contributions of many people at Cornell University and elsewhere. My thesis advisor, Christopher Batten, has been tremendously helpful with insightful advice and feedback in research ideas and writing in all of the projects presented in this thesis. Furthermore, Carl Friedrich Bolz-Tereick has been a very valuable mentor and collaborator for most of the work presented in this thesis.

The Pydgin project was lead by Derek Lockhart and was published at ISPASS 2015 [LIB15]. I led the meta-tracing JIT optimizations and co-led the RISC-V port [ILB16]. Additionally, Carl Friedrich Bolz-Tereick gave useful feedback for the paper and help with advanced optimization options of the RPython framework, Maciej Fijalkowski gave useful feedback for the paper, and Yunsup Lee helped towards the RISC-V port effort. Shreesha Srinath has been an early and extensive user of Pydgin and gave useful feedback. I additionally thank Sarah Mount for her work in using Pydgin for the Adapteva Epiphany instruction set. Even though Lockhart's thesis also covers Pydgin [Loc15], Chapter 3 emphasizes aspects not covered earlier, including the RISC-V port and self-modifying code support. Also importantly, it forms the foundation for the PydginFF project. I have led the PydginFF project, which was published at ISPASS 2016 [IB16]. I would like to thank Trevor Carlson for his feedback for improving the paper. The Mamba project was led by Shunning Jiang and was published at DAC 2018 [JIB18]. I co-led the meta-tracing JIT optimizations to achieve competitive performance compared to commercial Verilog simulations. I have included parts of Mamba work primarily because it demonstrates a complete picture of the power of meta-tracing JIT VMs towards accelerating agile hardware simulation at different modeling levels.

I have led the work of cross-layer workload characterization of meta-tracing JIT VMs and published this work at IISWC 2017 [IBTB17]. Carl Friedrich Bolz-Tereick helped with useful feedback and literature review and Alex Katz helped modifying and running benchmark scripts. Finally, Lin Cheng led cycle-level evaluation and parts of functional-level evaluation for the work presented in Chapter 7. Carl Friedrich Bolz-Tereick has helped with research direction and useful feedback for this project. Mohammad Dohadwala, Scott McKenzie, and Alvin Wijaya helped with various aspects of the earlier iterations of the project. Also thanks to Christopher Torng for leading the BRGTC2 chip tapeout and accommodating the addition of Bloom filter accelerators [TJAH+18].

This work was supported in part by NSF CRI Award #1512937, NSF SHF Award #1527065, NSF XPS Award #1337240, NSF CAREER Award #1149464, DARPA POSH Award #FA8650-18-2-7852, DARPA Young Faculty Award #N66001-12-1-4239, AFOSR YIP Award #FA9550-15-1-0194, and equipment, tool, and/or physical IP donations from Intel, Xilinx, Synopsys, Cadence, and ARM. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

CHAPTER 2
BACKGROUND ON META-TRACING JIT VMS

This chapter provides background on dynamic languages, just-in-time (JIT) compilation, types of JIT compilers, classical JIT optimization techniques, and partial evaluation. Furthermore, it provides details on the meta-tracing JIT technology, which is an essential part of this thesis. The first thrust of the thesis, accelerating hardware simulations with meta-tracing JIT VMs, exploits the features provided by meta-tracing in novel ways to accellerate architectural simulations. The second thrust, building hardware accelerators for meta-tracing JIT VMs, identifies shortcomings in meta-tracing and proposes software/hardware co-design techniques to address those. The RPython framework is the only production-grade implementation of a meta-tracing JIT framework, so this chapter also provides details on this framework.

2.1 Dynamic Languages

Languages such as C, C++, Java, Rust, and Go are all statically typed: the types of variables are statically declared by the programmer, or are statically inferred. Once determined, the type system ensures that the types of variables cannot be modified. Static type systems can encourage disciplined programming and enable sophisticated correctness checks of the program before any execution takes place. Also very importantly, static typing generally makes it more efficient to execute a program. In a program, operators may be overloaded across multiple data types (e.g., the + operator can act as integer addition or string concatenation) and unrelated classes might have members with identical lexical names, leading to ambiguity which semantics should be performed without static typing. Static typing allows compilers to precisely determine which concrete version of overloaded operators need to be used, and generate specialized code for this particular type. Static *and* strong typing (i.e., no reinterpretive type casting), for example in Java, Go, and Rust, can enable further compiler optimization opportunities (e.g., type-based alias analysis [CWZ90, DMM98]).

In contrast to statically typed languages, dynamically typed languages do not statically bind types to variables. As the program executes, a variable might contain data of a certain type, and later take on data of an unrelated type. While orthogonal to the typing system, many popular dynamically typed languages, also known as dynamic languages, have other common features: strong typing (not to be confused with static typing), lightweight syntax, managed memory, rich standard libraries,

```
 1 # Variables can contain objects of different types.
 2 a = 1; a = "foo"
 3
 4 # Unrelated classes that happen to both have methods called bar.
 5 class A:
 6   def bar(self):
 7     return 0
 8 class B:
 9   def bar(self):
10     return "bar"
11
12 # Lists can contain objects of unrelated types.
13 l = [ A(), B() ]
14 # Duck typing for unrelated types, prints "[0, 'bar']".
15 print [ x.bar() for x in l ]
16
17 # Reflection allows programmatically retrieving class object A using a string.
18 C = globals()["A"]
19 a = C()
20 # We can call the bar method again using reflection, prints 0.
21 print getattr( a, "bar" )()
22
23 # Reflection allows modifying the type A.
24 A.bar = lambda self : "new_value"
25
26 # The line from above, with objects constructed before we modified the type A,
27 # now prints "['new_value', 'bar']".
28 print [ x.bar() for x in l ]
```

Figure 2.1: Dynamic Language Features – Examples of features that dynamic languages typically provide.

support for dynamic code execution (e.g., the `eval` keyword in Python and JavaScript), interactive execution environments, and advanced reflection capabilities. The categorization of dynamic and static languages is not always clear cut, and languages are typically on a spectrum. For example, Java, a static language, has many dynamic features: it makes heavy use of dynamic method dispatch, uses managed memory, type inference, dynamic code execution, and reflection. Unfortunately, many of these powerful dynamic features mean that ahead-of-time- (AOT-) compilation will not result in meaningful performance improvements. An AOT compiler, without a priori knowledge of the types of variables, cannot generate efficient, type-specialized code. Because of this, dynamic languages typically use interpreters for execution.

Figure 2.1 shows some of the dynamic features that are typically present in dynamic languages through an example Python snippet. Line 2 shows that a variable `a` is allowed to contain values of different types throughout execution; line 13 shows that builtin types such as lists allow containing

Year:	2018	2018	2018	2018	2014	2013	2013
Ref:	[Cas18]	[TIO18]	[Car18]	[O'G18]	[Tre15]	[Lan13]	[tlp13]
Rank	**IEEE Spectrum**	**TIOBE**	**PYPL**	**RedMonk**	**Trendy Skills (best paying jobs)**	**LangPop**	**Trans. Lang. Pop. Idx.**
1	**Python**	Java	**Python**	**JavaScript**	**Python**	C	C
2	C++	C	Java	Java	Java	Java	Java
3	Java	**Python**	**JavaScript**	**Python**	C++	**PHP**	Objective-C
4	C	C++	C#	**PHP**	C	**JavaScript**	C++
5	C#	Visual Basic	**PHP**	C#	**JavaScript**	C++	Visual Basic
6	**PHP**	C#	C/C++	C++	HTML5	**Python**	**PHP**
7	**R**	**JavaScript**	**R**	CSS	XML	Shell	**Python**
8	**JavaScript**	**PHP**	Objective-C	**Ruby**	**PHP**	**Ruby**	C#
9	Go	SQL	Swift	C	C#	Objective-C	**Perl**
10	Assembly	Objective-C	**Matlab**	Objective-C	HTML	C#	**Ruby**

Table 2.1: Top Ten Most Popular Programming Languages – Adapted from [RO14] and updated. Using most up-to-date versions of the rankings as indicated by year. Languages in **bold** are dynamic languages.

objects of unrelated types; line 14 demonstrates duck typing, where the `bar` methods of unrelated objects would be called without the need to use inheritance; lines 18–21 show the power of reflection in programmatically accessing variables and types, constructing, and calling them; line 24 demonstrates monkey patching, where a different method `bar` gets attached to class `A`; and line 28 shows objects of type `A` that were instantiated before `A` has been modified are affected by the type modification.

Even though dynamic typing gives up static error checking through type checking, the powerful features of these languages while maintaining simplicity can help productivity. Productivity is a notoriously difficult and subjective metric to measure, and there have been a few empirical studies that suggest programmers prefer static typing in the interfaces of programs but dynamic in smaller, frequently changing code [SF14, HKR+14, DSF09]. Furthermore, dynamic languages have been increasing in popularity. Table 2.1 shows the top ten most popular languages across different rankings that compare the popularity of keywords in search engines, social media sites, and public code repositories such as GitHub. Even though the exact ordering is different, every ranking indicates that three to five of ten most popular languages are dynamic languages with Python, JavaScript, PHP, Ruby, and R consistently selected as the most popular dynamic languages.

2.2 Just-in-Time Compilation

Just-in-time- (JIT-) compilation (often also referred to as simply JIT or adaptive optimization), is an optimization technique with a long history [AFG+05, Ayc03]. As in Arnold et al. [AFG+05], it is useful to employ Rau's categorization of program representations [Rau78] to crisply define what JIT compilation is:

- *High-level representation (HLR)*: high-level source code, e.g., C++, Java, or Python. (For example, Figure 2.2)
- *Directly interpretable representation (DIR)*: an intermediate-level representation suitable for executing on an interpreter, e.g., Java bytecode or Python bytecode. (For example, Figure 2.3)
- *Directly executable representation (DER)*: binary code assembled for a particular ISA, e.g., x86 or RISC-V binary.

An interpreter is a piece of software that directly executes DIR, whereas a JIT compiler is a piece of software that translates DIR to DER. Some of the earliest JIT compilers had the primariy goal of reducing the memory usage. As the available memory of early computers was exteremely limited, and because DIR tended to be a much more compact representation than DER, there were two related techniques proposed to reduce the memory usage: mixed code [DP73, Daw73], where frequently executed parts of the code are statically compiled to DER and others were interpreted, and throw-away compiling [Bro76], where parts of a program are dynamically compiled on an as-needed basis and thrown away if memory is exhausted.

While earlier efforts on JITs were primarily driven as a space-saving measure for static languages, the poor performance of the interpreter for Smalltalk, an early dynamic language, made clear that the performance of interpreter-only execution was an issue. An important early work on JITs for dynamic languages was implemented in the VM for Smalltalk-80 [DS84]. Then, the Self project introduced many important advanced JIT techniques still used today: polymorphic inline caches [HCU91], on-stack replacement [HU94b], dynamic deoptimization [HCU92], selective compilation with multiple tiers of compilers [DA99], type prediction and splitting [CU91], and profile-directed inlining integrated with adaptive recompilation [HCU91].

Then, with the introduction of Java, JIT-optimized VMs became truly mainstream. The early versions of Java VMs did not have JIT compilers, resulting in very poor performance compared to

```python
def max(a, b):
  if a > b:
    return a
  else:
    return b
```

Figure 2.2: Example Python Code (HLR)

```
# Line 2:
 0 LOAD_FAST          0 (a)
 3 LOAD_FAST          1 (b)
 6 COMPARE_OP         4 (>)
 9 POP_JUMP_IF_FALSE 16
# Line 3:
12 LOAD_FAST          0 (a)
15 RETURN_VALUE
# Line 5:
16 LOAD_FAST          1 (b)
19 RETURN_VALUE
20 LOAD_CONST         0 (None)
23 RETURN_VALUE
```

Figure 2.3: Python Bytecode (DIR) – Python bytecode uses a stack machine and this code corresponds to Figure 2.2

```
w1_type = w1.type
w2_type = w2.type
guard(w1_type == IntObj)
guard(w2_type == IntObj)
i1 = w1.intval
i2 = w2.intval
res = i1 > i2
if res == True:
  retval = w1
else:
  retval = w2
```

Figure 2.4: JIT Pseudo-IR – The DER before assembly, corresponding to Figure 2.3.

```python
class Frame:
  def __init__(self, size):
    self.stack = [None] * size
    self.depth = 0
  def pushval(self, val):
    self.depth += 1
    self.stack[self.depth] = val
  def popval(self):
    self.depth -= 1
    return self.stack[self.depth + 1]

def interpret(bcs, frame):
  while True:
    bc = bcs[i]
    if bc == LOAD_FAST:
      idx = bcs[i + 1]
      frame.pushval(frame.stack[idx])
      i += 3
    elif bc == COMPARE_OP:
      cmpop = bcs[i + 1]
      w2 = frame.popval()
      w1 = frame.popval()
      if cmpop == 0:   # lt
        ...
      elif cmpop == 4: # gt
        if w1.type == IntObj and \
           w2.type == IntObj:
          i1 = w1.intval
          i2 = w2.intval
          frame.pushval(i1 > i2)
        elif w1.type == FloatObj and \
             w2.type == FloatObj:
          ...
        elif w1.type == UserInstanceObj:
          frame.pushval(w1.call('__gt__', w2))
        elif ...  # Other types.
      elif ...  # Other cmpops.
      i += 3
    elif ...  # Other bytecodes.
```

Figure 2.5: Simplified Python Interpreter

C and C++. Because Java is mostly a static language, AOT-compilation is possible (e.g., gcj [gcj18], Java 9 [Koz18], and Android ART [and]). However, some of the dynamic features of Java (e.g., dynamic class loading) cannot be AOT-compiled, so those are either not allowed or fall back to being executed on the interpreter and JIT. Since efficient AOT-compilation of Java is possible, the main purpose of using a JIT VM for Java is portability across multiple targets and security.

Over the last decade, JIT VMs have increased their mainstream adoption, but this time in the context of dynamic languages. Because types are dynamically determined, the best AOT compilation can achieve is to get rid of the DIR dispatch overhead, and this would only be marginally faster than execution on an interpreter. The AOT-compiled code would have to include the type dispatch logic and semantics of all types. Furthermore, dynamic languages typically allow programmatic creation and execution of new programs (e.g., Python and JavaScripts's `eval()`). Dynamically created programs, by definition, cannot be AOT compiled. Because of this, as opposed to Java JIT's goal of primarily providing portability, JITs for dynamic languages primarily aim to improve the performance. JIT VMs for dynamic languages are ubiquitous these days especially with the JavaScript engines of browsers (e.g., Google (Chrome) V8 [v8], Mozilla (Firefox) SpiderMonkey [spi], Webkit (Safari) JavaScriptCore [jav], and Microsoft (Edge) Chakra [cha]), but also for servers (e.g., HHVM for PHP [Ott18]), scientific computing (e.g., Julia [jul]), and general purpose programming (e.g., PyPy for Python [pypa], TruffleRuby for Ruby [trub], and LuaJIT for Lua [lua]).

Executing a dynamic language program on an interpreter has a number of overheads compared to running the same program in C. We will look at how JIT compilation can address some of these overheads:

- *DIR dispatch overhead*: An interpreter has to get the next available DIR instruction (e.g., bytecode), decode it, and dispatch to the interpreter code that implements the semantics of this instruction. This is shown in lines 15, 19, and 39 in Figure 2.5.
- *Type dispatch overhead*: In a dynamic language, since types are dynamically determined, the semantics of operations changes depending on the type. Lines 26, 31, and 34 in Figure 2.5 show an example of these overheads.
- *Semantic overhead*: The language may provide more semantically rich operators and data types compared to a lower level language. For example, integers in Python are semantically infinite bits wide. Even if a programmer does not need infinite-width integers, machine-width integers might not be available in the language or more cumbersome to use.

There are a number of design decisions regarding the construction of JIT VMs for dynamic languages. Figure 2.6 shows a typical architecture. The HLR gets compiled down to DIR (arrow 1), and DIR executes on an interpreter (arrow 2). The interpreter serves three purposes: it can execute DIR with low latency (but low peak performance as well), identify hot (frequently executed) parts

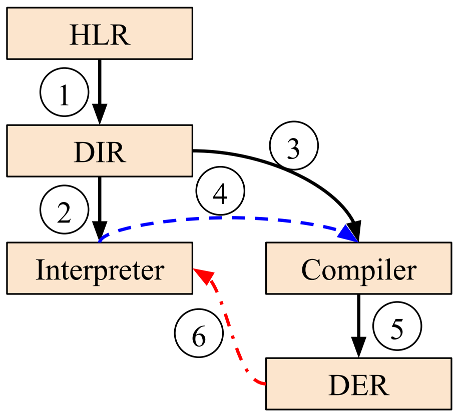


Figure 2.6: Typical JIT VM Architecture – Arrow 1: high-level representation (HLR) compiles to directly interpretable representation (DIR), 2: the DIR is interpreted using an interpreter, 3: hot DIR methods can be compiled using a JIT compiler, 4: for good performance, runtime type information also needs to be used by the compiler, 5: the compiler generates directly executable representation (DER), and 6: if the type assumptions in the DER are incorrect, the execution falls back to the interpreter using deoptimization.

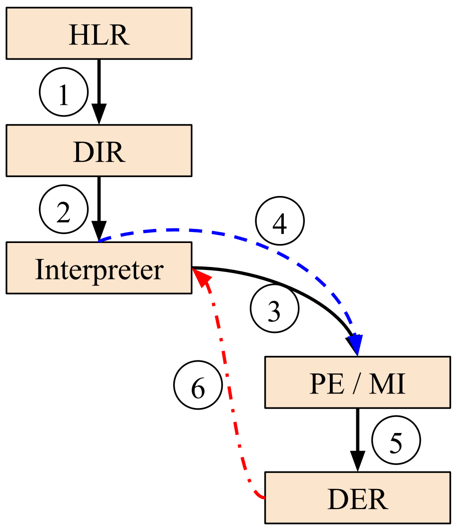


Figure 2.7: Meta-JIT VM Architecture – PE = partial evaluator, MI = meta-interpreter. Arrows 3 and 4: the interpreter itself is used to generate the DER along with runtime type information. The other arrows are the same as Figure 2.6

of the DIR, and gather runtime information such as types and constant values. Once a part of the DIR is determined to be hot enough, the (JIT) compiler compiles it to DER (arrows 3 and 5). Importantly, the compiler also uses the runtime information gathered by the interpreter (arrow 4). Without this information, the best a compiler can do is to get rid of the DIR dispatch overhead. Using these runtime type observations, the compiler can generate type-specialized DER. As there are no semantic guarantees of the type information to be stable, the DER generated also includes type checks. In case any of these type checks fail, the DER would not be able to handle the program semantics that it was not specialized for, and it deoptimizes to the interpreter (arrow 6). The interpreter can then continue execution, and compile again with the additional observed type.

2.2.1 JIT Design Decisions

Multi-Tier Compilation – While Figure 2.6 shows an architecture with an interpreter and a compiler, many of the state-of-the-art JIT VMs for dynamic languages employ multiple levels of compilation. For example, JavaScriptCore used in Safari uses an interpreter, a baseline non-optimizing compiler that does not use the type information to specialize, a second-tier optimizing compiler that uses the runtime types and values observed by the interpreter and baseline compiler, and finally a third-tier optimizing compiler that uses more aggressive and expensive compilation passes. If the DER produced by any of the optimizing compilers encounters a type that it is not optimized for, it falls back to the interpreter or DER produced by the baseline compiler [jav]. Other JavaScript VMs, such as V8, use similar multi-tier architectures. The chosen VM architecture aims to balance latency and peak performance. More aggressive compilation passes produce better code but require additional compilation time. As the VM does not have a priori information about how often a part of a DIR will be executed, it heuristically uses consecutively more aggressive compilation tiers in an effort to balance time spent in compilation and the performance of DER. One of the drawbacks of having an interpreter and multiple tiers of compilers is the engineering challenge of developing and maintaining two separate and complicated pieces of code.

Granularity of Compilation – Another design decision is the granularity of compilation for the DERs. The traditional approach is to compile DERs that correspond to HLR methods, and this approach is called a "method JIT". Besides methods, contemporary method JITs also have special support to compile regions smaller than methods such as loops, and larger than methods using method inlining and region-based compilation [Ott18, HHR95]. Method JITs maintain the control-flow graph in the DIR in the code that they compile. An alternative approach to method JITs is a tracing JIT, which captures and compiles a linear trace through the DIR [GES$^+$09]. Tracing JITs compile a linear DER, which contains guards for control-flow diversions. In case the control flow takes a different path than the linear DER, these guards would fail, and bridges, which are additional linear traces that attach to these failing guards may be compiled. One benefit of tracing JITs over method JITs is that certain compiler optimizations may be less expensive due to cheaper compiler optimization passes that do not have to deal with control flow diversions.

2.2.2 Classic JIT Optimizations and Techniques

There has been a number of JIT optimizations developed over the years tackling the dynamic language execution overheads. Below is a list of some of the most well known ones:

DIR Dispatch Overhead Removal – This is the first and easiest overhead to tackle. This is also the only optimization that AOT compilers for dynamic languages can implement. This optimization can be implemented by inlining the semantics of the DIR instructions.

```
w1_type = w1.type
w2_type = w2.type
if w1_type == IntObj:
  guard(w2_type == IntObj)
  i1 = w1.intval
  i2 = w2.intval
  res = i1 > i2
else:
  if w1_type == UserInstanceObj1:
    res = call(UserInstanceObj1_gt_fun, w1, w2)
  elif w1_type == UserInstanceObj2:
    res = call(UserInstanceObj2_gt_fun, w1, w2)
  else:
    gt_fun = call(get_fun, w1,  "__gt__")
    res = call(gt_fun, w1, w2)
if res == True:
  retval = w1
else:
  retval = w2
```

Figure 2.8: JIT Pseudo-IR with a Polymorphic Inline Cache (PIC) – Corresponds to the pseudo-IR from Figure 2.4, but after two additional user types have been encountered and optimized using a PIC.

Type Feedback – Even though dynamic languages allow different types at each DIR instruction, in practice, programmers only use a few types. Optimizing away the type dispatch overhead uses this observation. The first step towards optimizing this overhead is for the interpreter (or sometimes first-tier DER as in JavaScriptCore [jav]) to collect type information and feed this to the optimizing compiler. The optimizing compiler would use this information to generate specialized DER for the observed types. It will also include guards for checking if the types of variables match the types the DER was optimized for. This technique was first used in Self [HU94a].

Inline Caching – One of the optimizations that can be done by the JIT compiler using type feedback is inline caching. For example, in dynamic-language method call such as `foo.bar()`, the type of the receiver object (`foo`) often cannot be deduced statically. However, in practice, very few types might be observed at the call site. Normally, a type dispatch needs to be performed to find the concrete method that corresponds to the type of the receiver object. This optimization inlines into the DER the concrete method along with a type check to ensure the receiver object is indeed what it was expected. Figure 2.4 shows an example of what inline caching might look like in the DER. This example shows that the type for the `max` function arguments are observed to be `IntObj`. Using this observation, the semantics of the greater-than function (`__gt__`) for integer types have been inlined in the trace. Type checking is done using guards, allowing falling back to the interpreter in case different types are observed when executing the DER. Note that even though type feedback is often used to determine which types to inline cache, inline caching can also be used without type feedback but with type heuristics (using common paradigms in the language). This technique was first used in the Smalltalk system [DS84].

Polymorphic Inline Caches (PIC) – An extension of inline caching is polymorphic inline caches. While the vast majority of the call sites indeed have a single receiver object type (monomorphic), deoptimizing to the interpreter every time the inline cache is different can be expensive. This approach grows the inline caches for all the common receiver object types observed at each call site. Figure 2.8 shows an example PIC in the DER that corresponds to the earlier DER in Figure 2.4, after additional user-defined types have been encountered and DER recompiled. This particular example uses a polymorphic inline cache both for the builtin `IntObj` type as well as two different user-defined types. In case the argument is of a fourth type, this code will manually retrieve the `__gt__` function for the type, which can be expensive. This technique was first implemented in Self [HCU91].

On-Stack Replacement (OSR) – One issue of having multiple representations of the program, the DIR and DERs produced through multiple levels of compilation is that switching from one representation to another is difficult because each representation uses a different stack layout. This is especially a problem if the current DER receives a different type that what it was specialized for, and needs to deoptimize. It cannot simply return from the currently executing DER because there is intermediate state in the current stack frame that needs to be communicated to the interpreter so that it can continue execution from that point. Similarly, JIT compilation might need take place while the interpreter is in the middle of executing a method, especially important for loops. This technique retains enough meta-information and stack layouts of each representation to reproduce the stack layout for the target representation. This technique, too, was first demonstrated in Self [HU94b].

Semantic Overhead Optimizations – In addition to type specialization techniques, there are also a number of techniques that target the semantic overheads of dynamic languages. One example is the infinite-bitwidth integer types used in Python. Even though semantically, integers need to behave as if they have infinite bits, in practice, most values would fit in machine-bitwidth-sized integers. To efficiently handle this, the VM would actually use a machine-bitwidth-sized integer to represent the dynamic language integer, but check for overflows for any operation that might cause an overflow. In case of an overflow, it changes the representation of the integer to use multiple

```python
def f(n, x):
  if n == 0:
    return 1
  elif n % 2 == 1:
    return x * f(n - 1, x)
  else:
    return f(n / 2, x) ** 2

def f_13(x):
  return x * ((x * (x ** 2)) ** 2) ** 2
```

Figure 2.9: Partial Evaluation Example – The partial evaluation of `f` with `n == 13` yields `f_13`, and it has identical semantics to `f` as long as `n == 13`. Example adapted from [JG02].

words. Similarly, in Python, lists can contain any data type. However, in practice, it is quite common for programmers to put objects of the same type, such as a list of integers. The VM can then opportunistically use an integer array to hold the data. Besides saving memory, this also reduces the number of type checks for reads. For correctness, the VM checks for insertions into the list and changes the list to a generic object array if an object of a different type is being inserted. Note that these optimizations, unlike the optimizations targeting DIR dispatch and type dispatch, can be implemented for interpreter-only VMs.

2.3 Partial Evaluation

Partial evaluation is the specialization of a program for certain inputs. For example, as Figure 2.9 shows, it is possible to specialize a piece of code for a subset of its inputs. If we use a partial evaluator, *pe*, which is a piece of code that partially evaluates a program for certain inputs, we can specialize `f` like following: $pe(\texttt{f}, 13) \rightarrow$ `f_13`. `f_13` is a partially evaluated version of `f` where `n == 13`. The partially evaluated version can be significantly faster than the generic version, with the restriction that it only gives the correct answer if `n == 13`.

Partial evaluation can also be applied to interpreters of dynamic languages. This allows generating specialized code out of an interpreter. This is known as the *first Futamura projection* [Fut71, JG02]. There are a total of three Futamura projections:

1. Partial evaluation of an interpreter with a subset of the interpreter inputs (application source code and subset of inputs) yields specialized programs. For an interpreter that can evaluate

$interpreter(in1, in2) \rightarrow out$; partial evaluation can be used to yield $pe(interpreter, in1) \rightarrow sp$, where $sp(in2) \rightarrow out$.

2. Partial evaluation can be applied to other programs, including partial evaluators. Partial evaluation of a partial evaluator that has an interpreter as its input would generate a specialized partial evaluator for the language the interpreter is written for. A specialized partial evaluator is a compiler, as it takes in the application source code and generates a specialized program. $pe(pe, interpreter) \rightarrow compiler$, where $compiler(in1) \rightarrow sp$.

3. Taking this further, we can also generate a compiler generator by partially evaluating a partial evaluator with a partial evaluator as its input. The generated compiler generator will be able to generate compilers given interpreter source code as its input. $pe(pe, pe) \rightarrow compgen$, where $compgen(interpreter) \rightarrow compiler$.

The first Futamura projection is a very attractive observation, as it can potentially allow building JIT compilers from dynamic language interpreters *automatically*. It can eliminate the engineering problem of developing and maintaining an interpreter and a JIT compiler for a particular DIR. Furthermore, engineering effort can be concentrated on developing an aggressive partial evaluator that can be used to automatically build JIT compilers for multiple dynamic languages out of interpreters, which tend to require less engineering effort. While theoretically, partial evaluation can be applied to any program including interpreters, there are a number of practical challenges [JG02]. One problem is ensuring the termination of the partial evaluation algorithm. Partial evaluation might not terminate with certain patterns of recursive calls. To ensure termination, the algorithm would need to determine the bounds of partial evaluation, and needs to use the generic (unspecialized) version of the code. Secondly, determining which inputs are good candidates for specialization (e.g., specializing `in1` vs. `in2` vs. *both* `in1` and `in2` for `f`) is hard. The partial evaluator needs to avoid overspecialization (and the specialized code might not be used enough times to amortize the time it takes to specialize) and underspecialization (and the specialized code might not give sufficient speedups). Because of these challenges, fully automatic partial evaluators are not very practical.

The Truffle project was the first successful implementation of partial evaluation for automatically generating JIT VMs [WWH+17, WWW+13, WWS+12]. The Truffle framework allows writing abstract-syntax-tree (AST) interpreters for dynamic languages in Java. Each different AST node is represented using a Java class with the corresponding semantics of executing the AST node.

Additionally, these Java classes also store the observations about types or values that the interpreter writer deems to be usually constant. These observations are marked with a special Truffle annotation. Once the DIR methods or loops are determined to be hot enough, the Truffle framework performs partial evaluation of the interpreter-level AST node objects that correspond to the DIR method. The structure of the AST (i.e., the parent and children AST nodes) and any of the type and data observations that are specially marked would be used as part of partial evaluation (equivalent to `in1` in the example above). Similarly, to ensure termination of partial evaluation, the partial evaluation boundaries are also explicitly marked in the interpreter. This approach is used to automatically generate program specialization simply out of interpreters, albeit with a heavy use of partial-evaluation-related annotations. There are JIT VMs built with Truffle for many languages including JavaScript [WWS$^+$12], Ruby [Sea15], Python [WB13], R [SWHJ16], and LLVM IR (for C/C++ and x86 assembly) [RGM16], many of which are either the fastest or competitive with purpose-built JIT VMs for the language.

2.4 Meta-Tracing

A related approach to partial evaluation is meta tracing. Whereas a tracing JIT records and compiles a trace of the DIR as it is interpreted, meta-tracing records and compiles a trace of the *interpreter* as it executes the DIR. Meta-tracing can also be seen as a limited form of partial evaluation: while partial evaluation creates specialized code given the interpreter code and a subset of generally constant inputs (e.g., types), meta-tracing creates specialized code given a *subset* of the interpreter code (only the traced paths) and a subset of generally constant inputs. Similar to partial evaluation, meta tracing can be used to build a dynamic language JIT VM using the interpreter for the language. Because partial evaluation and meta-tracing both allow automatically building JIT VMs using the interpreter, they are sometimes collectively referred to as meta-JIT frameworks. The RPython framework is the only successful implementation of meta-tracing, and this framework serves as the foundation for the work in this thesis.

The RPython framework (RF) is a VM-generation framework that can also automatically generate a tracing-JIT compiler from the interpreter. Figure 2.10 shows the components of the framework and how these components interact with each other. Importantly, there is a translation-time and execution-time split, where the translation-time describes the events that take place

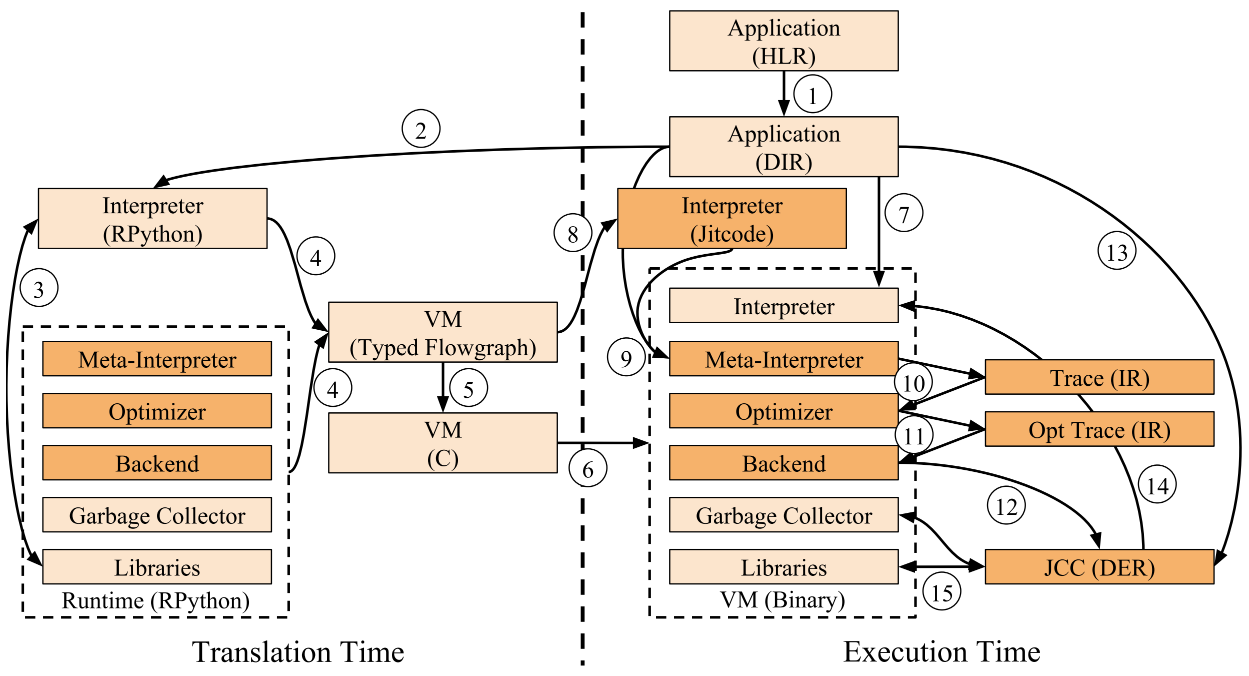


Figure 2.10: The RPython Framework – Darker blocks are needed to support meta-tracing JIT compilation.

once when building the VM, and execution-time describes the events that take place when the generated VM is used. Because the RPython name is used for the framework, language, runtime, and translation toolchain, the following discussion will refer to these components using their abbreviations for clarity even though these abbreviations are not widely used. At the simplest level, the RPython framework allows writing interpreters for a target dynamic language DIR (e.g., Python Bytecode) using the RPython language (RL). Arrow 1 in Figure 2.10 shows the HLR to DIR compilation. RL is a proper, typeable subset of Python. The RPython runtime (RR) itself is also written mostly in RL. Because RL is a subset of Python, the interpreter can be run on top of a Python VM, especially useful for debugging (arrow 2). Running the interpreter on a Python VM can make use of RR libraries (arrow 3), but garbage collection etc. will still be performed by the underlying Python VM. Running the interpreter on top of another VM is unsurprisingly very slow, but to get around that, RF includes the RPython translation toolchain (RTT). RTT can perform type inference over the entire interpreter and runtime (arrow 4). The typed graph is then converted to C and then compiled to binary using a C compiler to build the VM (arrows 5 and 6).

Once the VM is built, at execution time, the target application DIR can directly be run on the translated interpreter (arrow 7). The performance of this interpreter-only VM is similar to interpreter-only VMs written in lower level languages such as C. But the real benefit of RF is the ability to automatically generate a tracing-JIT compiler using the interpreter written in RL. In a typical tracing-JIT compiler, a trace of execution through the application bytecode is built and optimized. However, RF actually generates a *meta-tracing-JIT* compiler, where the trace is instead built by tracing through the *interpreter* as the interpreter executes the application bytecode. To achieve this, the runtime includes a meta-interpreter, an optimizer, and a backend to generate machine code, all written in RL. At translation time, in addition to generating C code, RF can use the typed flowgraphs to create jitcodes, which are memory-dense and statically typed representation of the interpreter semantics (arrow 8).

During execution time, as the application bytecode executes on the interpreter, frequently executed paths in the application are discovered. Once these paths are hot enough, the VM switches to using the meta-interpreter. The meta-interpreter executes the jitcodes, which in turn interprets the application bytecodes (arrow 9). As the meta-interpreter executes, it records a trace of the operations it performs (arrow 10). The generated trace is linear, but it includes guards that check for run-time conditions that would cause the execution to diverge from the recorded trace. Interpreters typically handle different run-time conditions in separate control flow paths (e.g., depending on the run-time type of a variable, perform the integer or string addition). So, generating the linear trace through the interpreter naturally contains the type-specialized paths. Using the guards and observed values of run-time variables, the optimizer can remove a large part of the trace, and then the DER of the optimized trace is generated (arrows 11 and 12). Once this occurs, the execution can directly happen on the JIT-compiled binary (arrow 13). Garbage collection and some parts of the VM libraries are not amenable to JIT-compilation, so there might be callbacks to these components from the JIT-compiled code (arrow 15). Finally, if any of the guards in the JIT-compiled code fails, the execution will fall back to the interpreter, where later another trace for the failing path might be traced and compiled if it is hot enough (arrow 14).

2.4.1 RPython Usage

To build high-performance meta-tracing JIT VMs using the RPython framework, there are a number of patterns and APIs that interpreters need to make use of. Below lists some of these important ones:

Translation Considerations – RL is a type-able restricted subset of the Python language. To determine the types, global type inference is performed over the interpreter and the RPython runtime.

Because of this, RL code cannot use dynamic typing (e.g., Line 2 in Figure 2.1 is not allowed). As a corollary duck typing (e.g., Line 15 in Figure 2.1) is also not allowed. However, (multiple) inheritance and polymorphism features of Python are retained in RL. Often, type inheritance is not sufficient to express VM writer's intent especially when it comes to type refinement. Unfortunately, the version 2 of Python that RL is based on does not support explicit type specification. To get around this, interpreter writers can use `assert isinstance(NewType, a)`, and RTT would treat this as downcasting. Similarly, lack of explicit type declaration can be a drawback in function arguments. Function overloading is allowed, but type inheritance tries to find a common ancestor if there are multiple calls with different types. Occasionally, there might not be a suitable common ancestor available. For those cases, interpreters can use the `@specialize.argtype` decorator, and this informs RTT to create multiple copies of the function with different types.

Basic JIT Hints – To take advantage of the meta-tracing JIT, at a minimum a `JitDriver` needs to be created, as shown in line 1 in Figure 2.11. The meta-tracing JIT traces the interpreter as it executes an iteration of an application-level loop. Because an application-level loop typically corresponds to multiple iterations of the interpreter-level loop, the framework needs to know which variables represent the location of the currently executing DIR instruction and when application-level loops jump back to the beginning. These are specified using *green* and *red* variables. The green variables represent the variables that indicate the DIR and the location into the DIR (e.g., PC, bytecode index, and the bytecodes themselves), and the red variables are the remaining variables. The hash of green variables serves as a unique identifier of DIR location, so this hash is used to find if the VM has already built JIT-compiled code at this particular location. The `JitDriver` object is used to mark the beginning of the interpreter-level loop using the `jit_merge_point` call (line 19) and application-level loop using the `can_enter_jit` call (line 47). With these hints alone, the meta-tracing JIT is able to trace through the application, but unfortunately this does not eliminate most of the overheads, as can be seen in the unoptimized pseudo-IR of the trace in Figure 2.12. Also note that the RPython framework supports multiple `JitDrivers` for different interpreter loops, useful for complex interpreter features implemented using loops, e.g., regular expressions, or for building mixed-language JIT VMs [BBT13].

Constant Promotion – Certain interpreter-level variables are constants at a particular point in the trace. A good example of this the DIR location (e.g., the PC). The constant promotion hint tells the optimizer that a particular variable can be treated as a constant at that particular point in the

```python
jd = JitDriver(greens=['i', 'bcs'], reds=['frame'])
class Frame:
  _virtualizable_ = ['stack[*]', 'depth', ...]
  def __init__(self, size):
    self.stack = [None] * size
    self.depth = 0
  def pushval(self, val):
    self.depth += 1
    self.stack[self.depth] = val
  def popval(self):
    self.depth -= 1
    return self.stack[self.depth + 1]

def interpret(bcs, frame):
  @elidable_promote
  def get_bc(bcs, i):
    return bcs[i]
  while True:
    jd.jit_merge_point(i=i, bcs=bcs, frame=frame)
    bc = get_bc(bcs, i)
    if bc == LOAD_FAST:
      idx = get_bc(bcs, i + 1)
      frame.pushval(frame.stack[idx])
      i += 3
    elif bc == COMPARE_OP:
      cmpop = get_bc(bcs, i + 1)
      w2 = frame.popval()
      w1 = frame.popval()
      if cmpop == 0:   # lt
        ...
      elif cmpop == 4: # gt
        if w1.type == IntObj and \
           w2.type == IntObj:
          i1 = w1.intval
          i2 = w2.intval
          frame.pushval(i1 > i2)
        elif w1.type == FloatObj and \
             w2.type == FloatObj:
          ...
        elif w1.type == UserInstanceObj:
          frame.pushval(w1.call('__gt__', w2))
        elif ...  # Other types.
      elif ...  # Other cmpops.
      i += 3
    elif bc == JUMP_ABSOLUTE:
      ...
      jd.can_enter_jit(i=i, bcs=bcs, frame=frame)
    elif ...  # Other bytecodes.
```

Figure 2.11: Simplified RPython Interpreter – Corresponds to the interpreter in Figure 2.5 but with meta-tracing hints.

```
i = 9
bc = bcs[i]
guard(bc != LOAD_FAST)
guard(bc == COMPARE_OP)
cmpop = bcs[i + 1]
stack = frame.stack
depth = frame.depth
w2 = stack[depth]
depth = depth - 1
frame.depth = depth
depth = frame.depth
w1 = stack[depth]
depth = depth - 1
frame.depth = depth
w1_type = w1.type
guard(w1_type == IntObj)
w2_type = w2.type
guard(w1_type == IntObj)
i1 = w1.intval
i2 = w2.intval
res = i1 > i2
stack = frame.stack
depth = frame.depth
depth = depth + 1
stack[depth] = res
frame.depth = depth
```

Figure 2.12: Unoptimized Meta-Trace

```
w1_type = w1.type
w2_type = w2.type
guard(w1_type == IntObj)
guard(w1_type == IntObj)
i1 = w1.intval
i2 = w2.intval
res = i1 > i2
guard(res == True)
retval = w1
```

Figure 2.13: Optimized Meta-Trace

```
a_type = a.type
b_type = b.type
guard(a_type == IntObj)
guard(b_type == IntObj)
i1 = a.intval
i2 = b.intval
i3 = i1 + i2
temp = new(IntObj)
temp.intval = i3
i4 = temp.intval
i5 = i4 / 2
c = new(IntObj)
c.intval = i5
```

Figure 2.14: Meta-Trace for `c=(a+b)/2` Without Escape Analysis

```
a_type = a.type
b_type = b.type
guard(a_type == IntObj)
guard(b_type == IntObj)
i1 = a.intval
i2 = b.intval
i3 = i1 + i2
i5 = i3 / 2
c = new(IntObj)
c.intval = i5
```

Figure 2.15: Meta-Trace for `c=(a+b)/2` With Escape Analysis

trace. Besides green variables, the careful use of this hint for unlikely-to-change variables (e.g., the shapes of application-level objects) can unlock further optimization opportunities. The variable that is constant promoted will have a guard in the trace to check if the run-time value is indeed equal to what was recorded in the trace, and this value would be constant propagated for the rest of the trace. The constant promotion hint is analogous to the constant inputs in a partial evaluation (as in `n` in Figure 2.9), and to the `PEFinal` keyword used in the Truffle framework. RPython also allows specifying immutable members of classes, analogous to the `const` members in C++. Immutable members also act similar to constant promotion, as the compiler can assume these values do not change once they are set.

Elidable Functions – Another powerful RPython JIT hint is the `@elidable` decorator. The interpreter often uses functions that return the same value given the same inputs. These are similar to pure functions, however, unlike pure functions, elidable functions can be used to optimize functions that have side effects but are idempotent. This can be useful for memoizing an expensive, but idempotent operation. As long as the arguments to the elidable function are constant (or constant promoted), the optimizer eliminates the call and replace it with the constant return value. This optimizer can further constant propagate the return value. Line 15 in Figure 2.11 shows the `@elidable_promote` decorator, which is a combined constant promotion and elidable function hint. This hint is used to eliminate the actual bytecode token given the bytecode list and an index. As the optimized trace in Figure 2.13 shows, this optimizes away not only the bytecode lookup logic, but also the bytecode dispatch and comparison operator dispatch logic.

Escape Analysis – One side effect of using the interpreter as the basis of JIT compilation is that the interpreter itself might introduce additional overheads. For example, a Python integer object is implemented using an interpreter object that has fields for the type and the raw integer value (`type` and `intval` in Figure 2.11 respectively). Furthermore, Python integers have immutable semantics, meaning any operation performed over them creates a new object instead of modify the existing one. Figure 2.14 shows a meta-trace that is generated from performing `c = (a + b) / 2`. Notice that lines 8–10 create a temporary integer object, store the temporary raw value to this object, and read this field back for further computation. This is because the interpreter had to create an `IntObj` object and perform boxing and unboxing operations. However, in this particular case, the temporary object is never read, and it does not escape the DER (be accessible to code outside the DER). Escape analysis in RPython identifies all of these unnecessary new object creations, boxing and unboxing operations, and removes them [BCF+11a]. After the escape analysis optimization, the code can be optimized like in Figure 2.15. If this trace is part of a larger trace where the escape analysis can determine `a`, `b`, and `c` are also used only as temporaries, the allocations and boxing operations can be optimized away for those objects too. The Truffle framework also performs a similar partial escape analysis to remove these overheads [SWM14].

Virtualizables – While escape analysis removes the overheads for the objects that are created in the DER that do not escape the DER, sometimes it is possible for an object to be visible outside the DER. Because of this possibility, partial escape analysis cannot optimize away redundant object creations, unboxing and boxing operations for those objects. For example, the DIR-level stack frame object can be created outside the DER and passed in to the DER, as seen between lines 2–12 in Figure 2.11. Here, if `stack` and `depth` are changed within the DER, the `Frame` object would need to be updated back for the rare case that code outside the DER also accesses these fields. The RPython framework supports virtualizables, which allows the removal of allocations and boxing operations while also maintaining the ability to access the `Frame` object from outside the DER, as seen in line 3 in Figure 2.11. Because operations over virtualizable objects are optimized away in the JIT-compiled code, the `Frame` object may contain stale versions of its members. For any code that accesses the `Frame` object outside the DER, the framework triggers a retrieval operation to find the most up-to-date values from the compiled DER's stack, similar to an on-stack replacement described in Section 2.2.2. This operation makes read and write operations to `Frame` object's fields cheaper in the DER (common case), at the expense of read and write operations outside the DER (uncommon case).

Quasi-Immutables – Immutable members of classes can unlock optimizations through constant propagations. However, there are are cases in an interpreter where certain class members can change extremely rarely. An example of this is the methods bound to a class in Python. While Python allows the addition or removal of methods to a class through monkey patching (e.g., in line 24 in Figure 2.1), most programs modify existing types only very rarely. RPython has a special hint to specify quasi-immutable members, and these are treated as immutables in the DER. RPython maintains a list of quasi-immutable members that a DER assumes to be immutable. Any write performed to these quasi-immutable members causes invalidation of all of the DERs that depend on that particular quasi-immutable. This is also known as external invalidation, and is also supported by Truffle using `Assumption` objects.

2.4.2 Meta-Tracing vs. Partial Evaluation

Truffle and RPython both achieve the goal of building JIT VMs with just an interpreter. This property is crucial for both thrusts of this thesis. In both approaches, the complicated JIT-generation framework is abstracted away. To make use of this framework, one has to write a relatively simpler interpreter. I demonstrate meta-JIT frameworks are excellent tools for building agile hardware simulators at various levels, including functional level, cycle level, and register-transfer level. Secondly, since meta-JIT frameworks allow building JIT VMs for multiple dynamic languages, it makes an excellent workload target to build hardware specialization. While this thesis could have used the Truffle framework, I have chosen the RPython framework for a number of reasons: (1) writing interpreters for RPython is slightly easier as it requires fewer hints [MD15]; (2) PyPy, the currently fastest implementation of Python, is written in RPython and Python has not had as many hardware acceleration proposals compared to JavaScript despite being a more popular language; (3) Python is arguably more general purpose than JavaScript; and (4) not all parts of the Truffle framework are open source (e.g., the Truffle JavaScript interpreter is not open source). Repeating the studies presented in this dissertation using the Truffle framework is a potential direction for future work.

PART I
ACCELERATING HARDWARE SIMULATIONS WITH META-TRACING JIT VMS

The first part of this thesis focuses on using the meta-tracing JIT technology to enable fast, accurate, and agile hardware simulation. Broadly, hardware systems are modeled at three different levels: functional-level (FL), cycle-level (CL), and register-transfer-level (RTL) modeling. FL models purely focus on the functional behavior of hardware and do not model time. FL models can be useful in initial design space exploration and techniques such as mixed-level modeling and sampling. CL models add timing simulation in addition to functional behavior. CL models do not model timing at perfect accuracy, but can still provide valuable insights as hardware designs are refined. RTL simulations accurately model register-level resources and time in terms of cycle count. While the increasing modeling detail can provide more accurate performance estimations, they also take much longer to develop and to simulate. Part I of the thesis explores using meta-tracing for all three modeling levels. Chapter 3 introduces Pydgin, where I present an approach to develop an interpreter using the RPython meta-tracing JIT framework for a functional model of a processor. Chapter 4 extends Pydgin to introduce JIT-compiled cycle-level models and embedding of the functional-level model inside another cycle-level simulator, allowing CL modeling through sampling and fast forwarding. Chapter 5 proposes novel meta-tracing JIT techniques to speed up PyMTL, a Python-based RTL modeling framework.

CHAPTER 3
PYDGIN: ACCELERATING FUNCTIONAL-LEVEL MODELING USING META-TRACING JIT VMS

This chapter explores the use of meta-tracing JIT VMs to accelerate functional-level (FL) hardware modeling. Pydgin provides a common framework for productively describing instruction set architectures (ISAs) using an RPython-based domain-specific language (DSL). Pydgin uses these instruction-set descriptions to create high-performance JIT-optimized instruction-set simulators by making use of RPython's meta-tracing JIT compiler. We implemented high-performance Pydgin ISSs for MIPS, ARM, and RISC-V ISAs.

While Pydgin has been created in collaboration with Lockhart [LIB15, Loc15], I have led the JIT optimizations to make Pydgin high-performance, and co-led the RISC-V port. Also importantly, this chapter highlights that meta-tracing JIT technology can be promising for functional-level hardware design, and is the starting point for my work on accelerating cycle-level hardware design in Chapter 4.

3.1 Introduction

Instruction-set simulators (ISSs) are used to functionally simulate instruction-set architecture (ISA) semantics. ISSs play an important role in aiding software development for experimental hardware targets that do not exist yet. ISSs can be used to help design brand new ISAs or ISA extensions for existing ISAs. ISSs can also be complemented with performance counters to aid in the initial design-space exploration of novel hardware/software interfaces.

Performance is one of the most important qualities for an ISS. High-performance ISSs allow real-world benchmarks (many minutes of simulated time) to be simulated in a reasonable amount of time (hours of simulation time). For the simplest ISS type, an interpretive ISS, typical simulation times are between 1 to 10 millions of instructions per second (MIPS) on a contemporary server-class host CPU. For a trillion-instruction-long benchmark, a typical length found in SPEC CINT2006, this would take many days of simulation time. Instructions in an interpretive ISS need to be fetched, decoded, and executed in program order. To improve the performance, a much more sophisticated technique called dynamic binary translation (DBT) is used by more advanced ISSs. In DBT, the target instructions are *dynamically* translated to host instructions and cached for future use. Whenever possible, these already-translated and cached host instructions are used to amortize much of the target instruction fetch and decode overhead. DBT-based ISSs typically achieve performance levels in the range of 100s of MIPS, lowering the simulation time to a few hours. QEMU is a widely used DBT-based ISS because of its high performance, which can achieve 1000 MIPS on certain benchmarks, or less than an hour of simulation time for a trillion-instruction-long benchmark [Bel05].

Productivity is another important quality for ISSs. A productive ISS allows *productive development* of new ISAs; *productive extension* for ISA specialization; and *productive custom instrumentation* to quantify the performance benefits of new ISAs and extensions. These productivity features are especially important for emerging open-source ISAs that encourage domain-specific extensions. For example, RISC-V is a new ISA that embraces the idea of specifying a minimalist standard ISA and encouraging users to use its numerous mechanisms for extensions for their domain-specific needs [WA17]. The users of RISC-V would likely need their ISS to be productive for extension and instrumentation while still needing the high-performance features to allow them to run real-world benchmarks. Bridging this productivity-performance gap requires agile simulation frameworks.

There has been previous ISSs that aim to achieve both productivity and performance. Highly productive ISSs typically use high-level architecture description languages (ADLs) to represent the ISA semantics [RABA04, ŽPM96, Pen11, QRM04, DQ06]. However, to achieve high performance, ISSs use low-level languages such as C, with a custom DBT, which requires in-depth and low-level knowledge of the DBT internals [May87, CK94, WR96, MZ04, TJ07, JT09]. To bridge this gap, previous research have focused on techniques to automatically translate high-level ADLs into a low-level language where the custom DBT can optimize the performance [PC11, WGFT13, QM03b]. However, these approaches tend to suffer because the ADLs are too close to low-level C, not very well supported, or not open source.

A similar productivity-performance tension exists in the programming languages community. Interpreters for highly productive dynamic languages (e.g., JavaScript and Python) need to be written in low-level C/C++ with very complicated custom just-in-time compilers (JITs). A notable exception to this is the PyPy project, the JIT-optimized interpreter for the Python language, which was written in a reduced typeable subset of Python, called RPython [BCF+11a, BCFR09, AACM07]. To translate the interpreter source written in RPython to a native binary, the PyPy community also developed the RPython translation toolchain. The RPython translation toolchain translates the interpreter source from the RPython language, adds a JIT, and generates C to be compiled with a standard C compiler. Chapter 2 provides an extensive introduction on RPython, meta-tracing JIT compilation, and related techniques. Pydgin is an ISS written in the RPython language, and uses the RPython translation toolchain to generate a fast JIT-optimized interpreter (JIT and DBT in the context of ISSs are very similar; we use both terms interchangeably in this chapter) from high-level architectural descriptions in Python [LIB15]. This unique development approach of Pydgin and the productivity features built into the Pydgin framework make this an ideal candidate to fill in the ISS productivity-performance gap.

In the remainder of this chapter, Section 3.2 provides an introduction to the Pydgin ADL and Pydgin framework, Section 3.3 explains the JIT annotations used in Pydgin to create a high-perfromance ISS, Section 3.4 provides examples of Pydgin productivity, Section 3.5 highlights the performance of Pydgin, and Section 3.7 concludes.

3.2 Pydgin ADL and Framework

Pydgin uses the RPython language as the basis of its embedded ADL. RPython has minimal restrictions compared to Python, such as dynamic typing not being allowed, however it still inherits many of the productivity features of full Python: simple syntax, automatic memory management, and a large standard library. Unlike other embedded ADL approaches that use a low-level host language such as C, architecture descriptions in Pydgin can make use of many of the high-level features of the Python host language. The ADL in Pydgin consists of architectural state, instruction encodings, and instruction semantics. The architectural state uses a plain Python class as shown in Figure 3.1. The Pydgin framework provides templates for various data structures such as memories and register files, and ISA implementers can either use these components directly or extend them to specialize for their architecture. For example, the RISC-V register file special-cases `x0` to always contain the value 0. Figure 3.2 shows how instruction encodings are defined in Pydgin using a user-friendly Python list of instruction name and a bitmask consisting of `0`, `1`, or `x` (for not care). Using this list, the Pydgin framework automatically generates an instruction decoder. When encountering a match, the decoder calls an RPython function with the name `execute_<inst_name>`. Figure 3.3 shows examples of execute functions that implement the instruction semantics. The execute functions are plain Python functions with two arguments: the state object and an instruction object that provides

```
class State( object ):
  _virtualizable_ = ['pc', 'num_insts']
  def __init__( self, memory, reset_addr=0x400 ):

    self.pc       = reset_addr
    self.rf       = RiscVRegisterFile()
    self.mem      = memory

    # optional state if floating point is enabled
    if ENABLE_FP:
      self.fp   = RiscVFPRegisterFile()
      self.fcsr = 0

    # used for instrumentation, not arch. visible
    self.num_insts = 0

    # state for custom instrumentation (number of
    # additions, number of misaligned memory operations,
    # list of all loops)
    self.num_adds       = 0
    self.num_misaligned = 0
    # a dict that returns 0 if the key is not found
    self.loop_dict      = DefaultDict(0)

```

Figure 3.1: Simplified RISC-V Architectural State Description – State for instrumentation can also be added here. The last three fields, `num_adds`, `num_aligned`, and `loops_dict` are examples of custom instrumentation state. Code that is highlighted in red is an advanced JIT annotation.

convenient access to various instruction fields such as the immediate values. The execute function can access and manipulate the fields in the state and instruction to implement the expected behavior.

The Pydgin framework provides the remaining ISA-independent components to complete the ISS definition. In addition to providing various templates for storage data structures, bit manipulation, system-call implementations, and ELF-file loading facilities, the framework most importantly includes the simulation loop. Figure 3.4 shows a simplified version of the simulation loop, which includes the usual fetch, decode, and execute calls. The Pydgin framework is designed so that ISA implementers and users would not need to modify the framework for most use cases. Since the ADL and the framework are written in RPython (a valid subset of Python), popular interpreters, debugging, and testing tools for Python can be used out of the box for Pydgin. This makes Pydgin development highly productive. Pydgin running on top of a Python interpreter is unsurprisingly very slow (around 100 thousand instructions per second), so this usage should be limited to testing and debugging with short programs.

```
encodings = [
  # ...
  [ 'xori',    'xxxxxxxxxxxxxxxxx100xxxxx0010011' ],
  [ 'ori',     'xxxxxxxxxxxxxxxxx110xxxxx0010011' ],
  [ 'andi',    'xxxxxxxxxxxxxxxxx111xxxxx0010011' ],
  [ 'slli',    '000000xxxxxxxxxxx001xxxxx0010011' ],
  [ 'srli',    '000000xxxxxxxxxxx101xxxxx0010011' ],
  [ 'srai',    '010000xxxxxxxxxxx101xxxxx0010011' ],
  [ 'add',     '0000000xxxxxxxxxx000xxxxx0110011' ],
  [ 'sub',     '0100000xxxxxxxxxx000xxxxx0110011' ],
  [ 'sll',     '0000000xxxxxxxxxx001xxxxx0110011' ],
  # ...

  # new instruction encodings can be added by for
  # example re-using custom2 primary opcode reserved
  # for extensions
  #['custom2', 'xxxxxxxxxxxxxxxxx000xxxxx1011011' ],
  [ 'gcd',     'xxxxxxxxxxxxxxxxx000xxxxx1011011' ],
]
```

Figure 3.2: RISC-V Instruction Encoding – Examples of instruction encodings showing some of the integer subset instructions. Encodings are defined using the instruction name and a bitmask consisting of `0`, `1`, and `x` (for not care). Reserved primary opcodes such as `custom2` can be used for instruction set extensions.

The performance benefits of Pydgin comes from using the RPython translation toolchain. RPython is a typeable subset of Python, and given an RPython source (e.g., PyPy or Pydgin), the RPython translation toolchain performs type inference, optimization, and code generation steps to produce a statically typed C-language translation of the interpreter. This C-language interpreter can be compiled using a standard C compiler to produce a native binary of an interpretive ISS. This interpretive version of Pydgin manages to reach 10 MIPS, which is much faster than interpreting Pydgin on top of a Python interpreter. However, the real strength of the RPython translation toolchain comes from its ability to couple a JIT compiler alongside the interpreter to be translated. The translated interpreter and the JIT compiler are compiled together to produce a DBT-based interpreter.

3.3 Pydgin JIT Generation and Optimizations

It is not immediately obvious that a JIT framework designed for general-purpose dynamic languages will be suitable for constructing fast instruction set simulators. In fact, a DBT-ISS

```
# add-immediate semantics
def execute_addi( s, inst ):
  s.rf[inst.rd] = s.rf[inst.rs1] + inst.i_imm
  s.pc += 4
  # count number of adds
  s.num_adds += 1

# store-word semantics
def execute_sw( s, inst ):
  addr = trim_xlen( s.rf[inst.rs1] + inst.s_imm )
  s.mem.write( addr, 4, trim_32( s.rf[inst.rs2] ) )
  s.pc += 4
  # count misaligned stores
  if addr % 4 != 0:
    s.num_misaligned += 1

# branch-equals semantics
def execute_beq( s, inst ):
  old_pc = s.pc
  if s.rf[inst.rs1] == s.rf[inst.rs2]:
    s.pc = trim_xlen( s.pc + inst.sb_imm )
    # record and count all executed loops
    if s.pc <= old_pc:
      s.loop_dict[(s.pc, old_pc)] += 1
  else:
    s.pc += 4

# extension example: greatest common divisor
def execute_gcd( s, inst ):
  a, b = s.rf[inst.rs1], s.rf[inst.rs2]
  while b:
    a, b = b, a%b
  s.rf[inst.rd] = a
  s.pc += 4
```

Figure 3.3: RISC-V Instruction Semantics – Examples of three RISC-V instructions: `addi`, `sw`, and `beq`; and `gcd` as an example of instruction-set extension implementing the greatest common divisor algorithm. Instructions in gray are examples of custom instrumentation that can be added and are optional.

generated by the RPython translation toolchain using only the basic JIT annotations results in only modest performance gains. This is because the JIT must often use worst-case assumptions about interpreter behavior. For example, the JIT must assume that functions might have side effects, variables are not constants, loop bounds might change, and object fields should be stored in memory. These worst-case assumptions reduce opportunities for JIT optimization and thus reduce the overall JIT performance.

```
jd = jit.JitDriver( greens = ['pc'], reds = ['state'],
                    virtualizables = ['state']  )
def instruction_set_interpreter( memory ):
  state = State( memory )
  while True:
    jd.jit_merge_point( state.pc, state )

    pc = jit.hint( state.pc, promote=True )
    mem = jit.hint( memory, promote=True )

    inst = memory.inst_read( pc, 4 ) # fetch
    execute = decode( inst )         # decode
    execute( state, inst )           # execute

    state.num_insts += 1

    if state.pc < pc:
      jd.can_enter_jit( state.pc, state )

```

Figure 3.4: Simplified Pydgin Interpreter Loop – Code that is highlighted in blue are the basic JIT annotations, and code that is highlighed in red are the advanced JIT annotations.

Existing work on the RPython translation toolchain has demonstrated the key to improving JIT performance is the careful insertion of advanced annotations that provide the JIT high-level hints about interpreter behavior [BCFR09, BCF+11a]. We use a similar technique by adding annotations to the Pydgin framework specifically chosen to provide ISS-specific hints. Most of these advanced JIT annotations are completely self-contained within the Pydgin framework itself. Annotations encapsulated in this way can be leveraged across any instruction set specified using the Pydgin embedded-ADL without any manual customization of instruction semantics by the user. Figure 3.4 shows a simplified version of the Pydgin interpreter with several of these advanced JIT annotations highlighted.

We use several applications from SPEC CINT2006 compiled for the ARMv5 ISA to demonstrate the impact of six advanced JIT annotations key to producing high-performance DBT-ISSs with the RPython translation toolchain. These advanced annotations include: (1) elidable instruction fetch; (2) elidable decode; (3) constant promotion of memory and PC; (4) word-based target memory; (5) loop unrolling in instruction semantics; and (6) virtualizable PC. Figure 3.8 shows the speedups achieved as these advanced JIT annotations are gradually added to the Pydgin framework. Speedups are normalized against a Pydgin ARMv5 DBT-ISS using only basic JIT annotations. Figures 3.6

```python
class Memory( object ):
  _immutable_fields_ = [ 'pages[*]' ]
  def __init__( self, size=2**10 ):
    self.size  = size << 2
    self.data  = [0] * self.size
    self.pages = [ Page() for i in range( num_pages ) ]

  def read( self, start_addr, num_bytes ):
    word = start_addr >> 2
    byte = start_addr &  0b11
    if   num_bytes == 4:
      return self.data[ word ]
    elif num_bytes == 2:
      mask = 0xFFFF << (byte * 8)
      return (self.data[ word ] & mask) >> (byte * 8)
    # ... handle single byte read ...

  def inst_read( self, start_addr, num_bytes ):
    return self._inst_read( start_addr, num_bytes,
                            self.get_page( start_addr ).version )

  @elidable
  def _inst_read( self, start_addr, num_bytes , version ):
    # The least significant bit of version indicates instruction page.
    if version & 1 == 0:
      self.get_page( start_addr).version += 1
    return self.data[ start_addr >> 2 ]

  @elidable
  def get_page( self, addr ):
    return self.pages[ addr >> 8 ]

  def write( self, start_addr, num_bytes, value )
    # ... handle the write ...
    # Make the page a data page if it was an instruction page previously.
    page = self.get_page( start_addr )
    if page.version & 1 == 1:
      page.version += 1

  # ... rest of memory methods ...

class Page( object ):
  _immutable_fields_ = [ 'version?']
  def __init__( self ):
    self.version = 0
```

Figure 3.5: Simplified Pydgin `Memory` and `Page` Classes – Notice that for performance reasons, instructions are loaded using the `inst_read` method, while remaining data loads use the `read` method. Code that are highlighed in red are advanced JIT annotations.

```
# Byte accesses for instruction fetch
i1 = getarrayitem_gc(p6, 33259)
i2 = int_lshift(i1, 8)
# ... 8 more JIT IR nodes ...

# Decode function call
p1 = call(decode, i3)
guard_no_exception()
i4 = getfield_gc_pure(p1)
guard_value(i4, 4289648)

# Accessing instruction fields
i5 = int_rshift(i3, 28)
guard_value(i5, 14)
i6 = int_rshift(i3, 25)
i7 = int_and(i6, 1)
i8 = int_is_true(i7)
# ... 11 more JIT IR nodes ...

# Read from regfile
i10 = getarrayitem_gc(p2, i9)

# Register offset calculation
i11 = int_and(i10, 0xffffff)
i12 = int_rshift(i3, 16)
i13 = int_and(i12, 15)
i14 = int_eq(i13, 15)
guard_false(i14)

# Read from regfile
i15 = getarrayitem_gc(p2, i13)
# Addressing mode
i15 = int_rshift(i3, 23)
i16 = int_and(i15, 1)
i17 = int_is_true(i16)
# ... 13 more JIT IR nodes ...

# Access mem with byte reads
i19 = getarrayitem_gc(p6, i18)
i20 = int_lshift(i19, 8)
i22 = int_add(i21, 2)
# ... 13 more JIT IR nodes ...

# Write result to regfile
setarrayitem_gc(p2, i23, i24)

# Update PC
i25 = getarrayitem_gc(p2, 15)
i26 = int_add(i25, 4)
setarrayitem_gc(p2, 15, i26)
i27 = getarrayitem_gc(p2, 15)
i28 = int_lt(i27, 33256)
guard_false(i28)
guard_value(i27, 33260)

# Update cycle count
i30 = int_add(i29, 1)
setfield_gc(p0, i30)
```

Figure 3.6: Unoptimized JIT IR for ARMv5 `LDR` Instruction – When provided with only basic JIT annotations, the meta-tracing JIT compiler will translate the `LDR` instruction into 79 JIT IR nodes.

```
i1 = getarrayitem_gc(p2, 0)     # register file read
i2 = int_add(i1, i8)            # address computation
i3 = int_and(i2, 0xffffffff)    # bit masking
i4 = int_rshift(i3, 2)          # word index
i5 = int_and(i3, 0x00000003)    # bit masking
i6 = getarrayitem_gc(p1, i4)    # memory access
i7 = int_add(i8, 1)             # update cycle count
```

Figure 3.7: Optimized JIT IR for ARMv5 `LDR` Instruction – Pydgin's advanced JIT annotations enable the meta-tracing JIT compiler to optimize the `LDR` instruction to just seven JIT IR nodes.

and 3.7 concretely illustrate how the introduction of these advanced JIT annotations reduce the JIT IR generated for a single `LDR` instruction from 79 IR nodes down to only 7 IR nodes. In the rest of

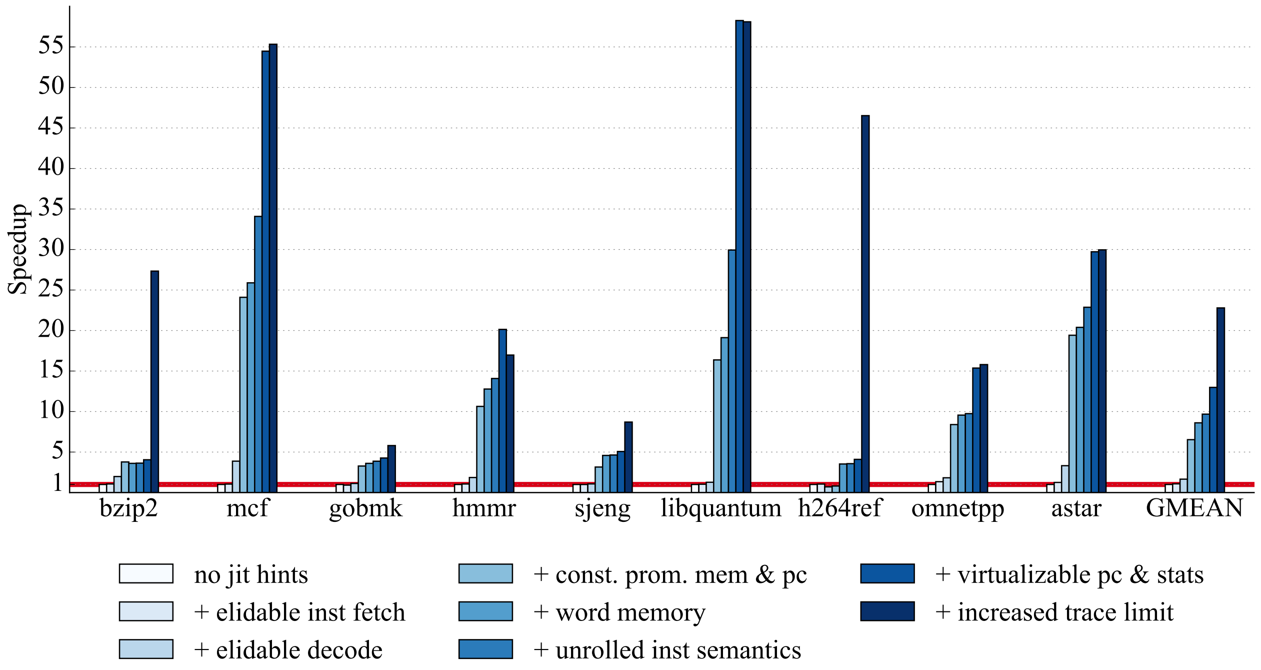


Figure 3.8: Impact of JIT Annotations – Including advanced annotations in the RPython interpreter allows our generated ISS to perform more aggressive JIT optimizations. However, the benefits of these optimizations varies from benchmark to benchmark. Above we show how incrementally combining several advanced JIT annotations impacts ISS performance when executing several SPEC CINT2006 benchmarks. Speedups are normalized against a Pydgin ARMv5 DBT-ISS using only basic JIT annotations.

this section, we describe how each advanced annotation specifically contributes to this reduction in JIT IR nodes and enables the application speedups shown in Figure 3.8.

Elidable Instruction Fetch – RPython allows functions to be marked *trace elidable* using the `@elidable` decorator. This annotation guarantees a function will not have any side effects and therefore will always return the same result if given the same arguments. If the JIT can determine that the arguments to a trace elidable function are constant, then the JIT can use constant folding to replace the function with its result and a series of guards to verify that the arguments have not changed. When executing programs without self-modifying code (see Section 3.3.1 for changes to allow self-modifying code), the Pydgin ISS benefits from marking instruction fetches as trace elidable since the JIT can then assume the same instruction bits will always be returned for a given PC value. While this annotation, seen on line 22 in Figure 3.5, can potentially eliminate 10 JIT IR nodes on lines 1–4 in Figure 3.6, it shows negligible performance benefit in Figure 3.8. This is because the benefits of elidable instruction fetch are not realized until combined with other symbiotic annotations like elidable decode.

Elidable Decode – Previous work has shown efficient instruction decoding is one of the more challenging aspects of designing fast ISSs [KA01, QM03a, FMP13]. Instruction decoding interprets the bits of a fetched instruction in order to determine which execution function should be used to properly emulate the instruction's semantics. In Pydgin, marking decode as *trace elidable* allows the JIT to optimize away all of the decode logic since a given set of instruction bits will always map to the same execution function. Elidable decode can potentially eliminate 20 JIT IR nodes on lines 6–18 in Figure 3.6. The combination of elidable instruction fetch and elidable decode shows the first performance increase for many applications in Figure 3.8.

Constant Promotion of PC and Target Memory – By default, the JIT cannot assume that the pointers to the PC and the target memory within the interpreter are constant, and this results in expensive and potentially unnecessary pointer dereferences. *Constant promotion* is a technique that converts a variable in the JIT IR into a constant plus a guard, and this in turn greatly increases opportunities for constant folding. The constant promotion annotations can be seen on lines 8–9 in Figure 3.4. Constant promotion of the PC and target memory is critical for realizing the benefits of the elidable instruction fetch and elidable decode optimizations mentioned above. When all three optimizations are combined, the entire fetch and decode logic (i.e., lines 1–18 in Figure 3.6) can truly be removed from the optimized trace. Figure 3.8 shows how all three optimizations work together to increase performance by 5× on average and up to 25× on *429.mcf*. Only *464.h264ref* has shown no performance improvements up to this point.

Word-Based Target Memory – Because modern processors have byte-addressable memories the most intuitive representation of this target memory is a byte container, analogous to a `char` array in C. However, the common case for most user programs is to use full 32-bit word accesses rather than byte accesses. This results in additional access overheads in the interpreter for the majority of load and store instructions. As an alternative, we represent the target memory using a word container. While this incurs additional byte masking overheads for sub-word accesses, it makes full word accesses significantly cheaper and thus improves performance of the common case. Lines 8–16 in Figure 3.5 illustrates our target memory data structure which is able to transform the multiple memory accesses and 16 JIT IR nodes in lines 38–42 of Figure 3.6 into the single memory access on line 6 of Figure 3.7. The number and kind of memory accesses performed influence the benefits of this optimization. In Figure 3.8 most applications see a small benefit, outliers include

401.bzip2 which experiences a small performance degradation and *464.h264ref* which receives a large performance improvement.

Loop Unrolling in Instruction Semantics – The RPython toolchain conservatively avoids inlining function calls that contain loops since these loops often have different bounds for each function invocation. A tracing JIT attempting to unroll and optimize such loops will generally encounter a high number of guard failures, resulting in significant degradation of JIT performance. The `stm` and `ldm` instructions of the ARMv5 ISA use loops in the instruction semantics to iterate through a register bitmask and push or pop specified registers to the stack. Annotating these loops with the `@unroll_safe` decorator allows the JIT to assume that these loops have static bounds and can safely be unrolled. One drawback of this optimization is that it is specific to the ARMv5 ISA and currently requires modifying the actual instruction semantics, although we believe this requirement can be removed in future versions of Pydgin. The majority of applications in Figure 3.8 see only a minor improvement from this optimization, however, both *462.libquantum* and *429.mcf* receive a significant improvement from this optimization suggesting that they both include a considerable amount of stack manipulation.

Virtualizable PC and Statistics – State variables in the interpreter that change frequently during program execution (e.g., the PC and statistics counters) incur considerable execution overhead because the JIT conservatively implements object member access using relatively expensive loads and stores. To address this limitation, RPython allows some variables to be annotated as *virtualizable*. Virtualizable variables can be stored in registers and updated locally within an optimized JIT trace without loads and stores. Memory accesses that are needed to keep the object state synchronized between interpreted and JIT-compiled execution is performed only when entering and exiting a JIT trace. The virtualizable annotation (line 2 in Figure 3.1 and line 2 in Figure 3.4) is able to eliminate lines 47–58 from Figure 3.6 resulting in an almost 2$\times$ performance improvement for *429.mcf* and *462.libquantum*. Note that even greater performance improvements can potentially be had by also making the register file virtualizable, however, a bug in the RPython translation toolchain prevented us from evaluating this optimization.

Increased Trace Limit – Unlike a method JIT that compiles a block of statically defined application code (typically an application method), tracing JITs build a trace of dynamically executed code. This means that the size of the trace built by a tracing JIT can be unbounded. The JIT cannot know a priori how long a trace will be, and if it is even bounded before tracing is completed. If a trace is extremely long, it will create significant memory and garbage collection pressure, and can result in optimizations to take a considerably long time, even with near-linear-time optimization algorithms. Furthermore, there is no guarantee how often a particular JIT trace will be utilized once it is compiled. After compilation, if the control flow diverges from the JIT-compiled trace, the remainder of the trace cannot be used. Given a longer trace usually has more guards, the odds of a guard failure due to divergent control flow gets higher as the trace grows longer. Due to these reasons, tracing JITs typically have a trace limit, where they abort building the trace if this limit is reached. The default trace limit in the RPython framework is tuned for dynamic languages such as Python, and balances giving up tracing too easily and unnecessarily compiling long traces that might not be amortized. However, we observed that the default heuristic of 6000 maximum IR nodes in the trace is not optimal for Pydgin. In particular, two of the benchmarks, *401.bzip2* and *464.h264ref* had a high number of trace aborts due to the trace size exceeding the trace limit. This is likely due to unrolling optimizations performed by C/C++ compilers, increasing the amount of code that needs to be traced. We swept the trace from the default 6000 to 800,000 and observed an $11\times$ performance boost in *464.h264ref* once the trace limit reaches 48,000 or more, and a $7\times$ boost in *401.bzip2* once the trace limit reaches 96,000 or more. We use 400,000 as the new trace limit, and as Figure 3.8 shows, only one benchmark (*456.hmmer*) had a modest slowdown with this trace limit.

3.3.1 Self-Modifying Code Support

While self-modifying code can be rare in C/C++ applications, JIT VMs work by generating optimized code and writing this to memory, and then execute this code. In addition, the tracing JIT in RPython actually modifies the compiled code to patch guards to compiled bridges instead of jumping to the blackhole interpreter. As JIT VMs are an important workload in general and to this thesis in particular, supporting self-modifying code on Pydgin is an important goal to allow PyPy to run on Pydgin.

Unfortunately, self-modifying code breaks the assumptions made in the *elidable instruction fetch* optimization. This optimization assumes at a given address, always the same value has to be returned. But if the application running on Pydgin modifies the data that corresponds to an executable instruction, the elidable instruction fetch would return the stale instruction. It is possible to disable this optimization, however the performance impact could make this prohibitive. To support high-performance execution of self-modifying code, we use a *page versioning* technique. We define a `Page` class (lines 42–45 in Figure 3.5) that has a `version` field. Instruction reads perform a call to `get_page` to find the corresponding `Page` object (line 20). The `get_page` method is *elidable* since the page object for a given address is the same (lines 29–31). In Pydgin, we support two types of pages: instruction and data. We encode instruction and data pages with odd and even version numbers respectively. In `_inst_read`, the version of the corresponding page is checked to see if it is a data or instruction page (line 25). If it is a data page, the version number is incremented (line 26), and the data can be returned (line 27). Because `_inst_read` is also *elidable*, the subsequent calls to this function with this address with the latest version can be optimized away.

To support self-modifying code, stores have to be special cased. In the `write` method, we perform a similar check on the corresponding page of the write address. As lines 36–38 show, if the page was an instruction page (odd), it has to be incremented by one to convert it into a data page. This means that subsequent calls to `_inst_read` will use a different value for `version`, and since there will not be a memoized value with the new version argument, the body of `_inst_read` has to be performed again. However, this is not enough, as changing the version only affects future tracing. Previously JIT-compiled code also needs to be invalidated in case an instruction, which was used to build the JIT-compiled trace, has since been modified. To achieve this, the RPython framework provides the *quasi-immutable* hint to be specified for a class. This can be seen in line 43, where the question mark indicates that the `version` field is a quasi-immutable member of `Page`. The quasi-immutable hint tells the JIT that the variable does not change very often, but is still allowed to change. Using this hint, the JIT compiler can remove the reads to `version` by employing a *write barrier* to the writers of `version`. A write barrier is a check in software that the VM inserts to all of the writers of the field. If this check is triggered, the VM invalidates all of the JIT-compiled traces that used that particular quasi-immutable variable. This effectively removes any lookup to page or the version in the JIT-traces for the common case. However, this requires additional overheads in the JIT-compiled trace for store instructions: the page is loaded and a guard is inserted to ensure that it is writing to a data page. Unlike Pydgin, QEMU uses a page-based translation scheme using the host pages, and to support self-modifying code, the translated code is invalidated if there are any writes to a translated page [Bel05]. So, QEMU directly leverages the host memory-management unit to find when the code is self-modifying, catch the appropriate exception, and avoid additional overheads in instruction fetch or store instructions. Using the host pages in this way is currently not possible using the RPython framework.

We have successfully integrated self-modifying code support into Pydgin, which allowed us to efficiently run JIT VMs on Pydgin, including PyPy.

3.4 Pydgin Productivity

Pydgin uses the RPython subset of standard Python in its ADL, and can use off-the-shelf Python testing and debugging tools. This makes developing implementations for new ISAs productive. Even though it is hard to quantify these productivity benefits, as a rough anecdote, Lockhart and I added RISC-V support to Pydgin over the course of nine days, three of which were during a busy conference. Implementing a simple (and slow) ISS in this duration is possible, however implementing one that achieves 100+ MIPS of performance is a testament to the productivity and performance features of Pydgin.

Facilitating domain-specific extensions to the existing ISA has become more commonplace with RISC-V. These extensions require adding the encoding and semantics of the new instructions to the ISS, and the software stack to take advantage of these new instructions. Pydgin makes the ISS-side modifications productive. Line 18 in Figure 3.2 shows an example of adding a new domain-specific greatest common divisor (`gcd`) instruction by reusing the `custom2` primary opcode reserved for extensions such as this. Lines 26–32 in Figure 3.3 show adding the semantics for the `gcd` instruction. Note that this particular example does not require any new architectural state to be added.

Adding custom instrumentation productively to the ISS is an important feature of Pydgin. Hardware and software designers are often interested in how well a new ISA or instruction-set extension performs running realistic benchmarks. Software designers are interested in knowing how often these new instructions are used to ensure the software stack is fully making use of the new features. Hardware designers are interested in knowing how software tends to use these new instructions, and what are the common arguments to a new functional unit, in order to design optimized hardware for the common case. Custom instrumentation in Pydgin often involves adding a new state variable as shown between Lines 14–23 in Figure 3.1. Here, we illustrate adding counters or more sophisticated data structures to count number of executed instructions, number of executed addition instructions, number of misaligned memory operations, and a histogram of all

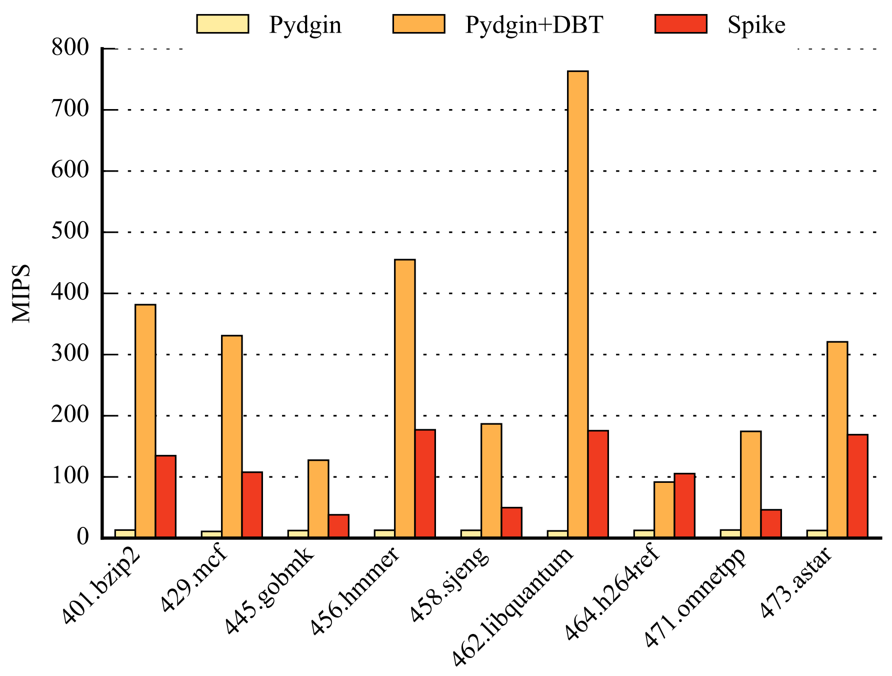


Figure 3.9: Performance Results – Pydgin without DBT, Pydgin with DBT, and Spike running SPEC CINT2006 benchmarks.

loops in the program. Figure 3.3 shows (in gray) how the new custom instrumentation can be added directly in the instruction semantics with only a few lines of code.

Productive development, extensibility, and instrumentation in Pydgin are complemented with the performance benefits of JIT compilation. Most of the JIT annotations necessary to use the full power of the RPython JIT are in the Pydgin framework, so the additions of the ADL automatically make use of these. The additional instrumentation code will automatically be JIT compiled. However, depending on the computational intensiveness and frequency of the instrumentation code, there will be a graceful degradation in performance.

3.5 Pydgin Performance

We evaluated the performance of Pydgin, with and without enabling the JIT, on SPEC CINT2006 benchmarks. We used a Newlib-based GCC 4.9.2 RISC-V cross-compiler to compile the benchmarks. We used system-call emulation in Pydgin to implement the system calls. Some of the SPEC CINT2006 benchmarks (`400.perlbench`, `403.gcc`, and `483.xalancbmk`) did not compile due to limited system-call and standard C-library support in Newlib, so we omitted these. Figure 3.9 shows the performance results. As expected, Pydgin without JIT achieves a relatively low performance of 10 MIPS across the benchmarks. The performance is fairly consistent regardless of the benchmark, and this is a characteristic of interpreter-based ISSs. Turning on the JIT in Pydgin increases the performance to 90–750 MIPS range. This performance point is high enough to allow running the reference datasets of these benchmarks (trillions of instructions) over a few hours of simulation time. Different patterns in the target code are responsible for the highly varied performance, and this is a characteristic of DBT-based ISSs.

To compare the performance of Pydgin to other ISSs for RISC-V, Figure 3.9 shows the performance of Spike, the reference RISC-V ISS. Despite being an interpreter-based ISS, Spike manages to achieve 40–200 MIPS, much faster than non-DBT Pydgin. This is because Spike is heavily optimized to be a very fast interpretive ISS, and employs some optimization techniques that are typical of DBT-based interpreters. Examples of these optimizations include a software instruction cache that stores pre-decoded instructions, a software TLB, and an unrolled PC-indexed interpreter loop to improve host branch prediction. These optimizations also cause the performance to be similarly varied.

At the time of development, the QEMU [Bel05] port of RISC-V was not available. We anticipate that the QEMU port is faster than Pydgin, however the fact that an up-to-date QEMU RISC-V port was not available for a prolonged time is indicative of the lack of productivity in performance-oriented ISSs. Pydgin's strength lies in combining productivity and performance.

3.6 Related Work

A considerable amount of prior work exists on improving the performance of instruction set simulators through dynamic optimization. Foundational work on simulators leveraging dynamic binary translation (DBT) provided significant performance benefits over traditional interpretive simulation [May87, CK94, WR96, MZ04]. These performance benefits have been further enhanced by optimizations that reduce overheads and improve code generation quality of JIT compilation [TJ07, JT09, LCL+11]. Current state-of-the-art ISSs incorporate parallel JIT-compilation task farms [BFT11], multicore simulation with JIT-compilation [QDZ06, ABvK+11], or both [KBF+12]. These approaches generally require hand modification of the underlying DBT engine in order to achieve good performance for user-introduced instruction extensions.

In addition, significant research has been spent on improving the usability and retargetability of ISSs, particularly in the domain of application-specific instruction-set processor (ASIP) toolflows. Numerous frameworks have proposed using a high-level architectural description language (ADL) to generate software development artifacts such as cross compilers [HSK+04,CHL+04, CHL+05, ARB+05, FKSB06, BEK07] and software decoders [KA01, QM03a, FMP13]. Instruction set simulators generated using an ADL-driven approach [ARB+05, QRM04, QM05], or even from definitions parsed directly from an ISA manual [BHJ+11], provide considerable productivity benefits but suffer from poor performance when using a purely interpretive implementation strategy. ADL-generated ISSs have also been proposed that incorporate various JIT-compilation strategies, including just-in-time cache-compiled (JIT-CCS) [NBS+02, BNH+04], instruction-set compiled (ISCS) [RDM06, RBMD03, RMD03], hybrid-compiled [RMD09, RD03], dynamic-compiled [QDZ06, BFKR09, PKHK11], multicore and distributed dynamic-compiled [DQ06], and parallel DBT [WGFT13].

Penry et al. introduced the *orthogonal-specification principle* as an approach to functional simulator design that proposes separating simulator specification from simulator implementation [Pen11, PC11]. This work is very much in the same spirit as Pydgin, which aims to separate JIT implementation details from architecture implementation descriptions by leveraging the RPython translation toolchain. RPython has previously been used for emulating hardware in the PyGirl project [BV09]. PyGirl is a whole-system VM (WSVM) that emulates the processor, peripherals, and timing-behavior of the Game Boy and had no JIT, whereas our work focuses on JIT-enabled, timing-agnostic instruction-set simulators.

3.7 Conclusion

Pydgin demonstrates meta-tracing JIT can be used to build high-performance instruction-set simulators (ISS), approaching the performance of purpose-built dynamic-binary-translation-based ISSs such as QEMU. Pydgin can achieve high performance while maintaining pseudo-code-like architecture descriptions. Achieving good performance using the RPython framework requires the insertion of hints at the interpreter level. However, many of these hints are inserted at the Pydgin framework level, so architecture implementers do not need to worry about VM-level hints. Pydgin can be especially productive from the perspective of adding additional instructions or instrumentation. Chapter 4 shows how I have used the Pydgin framework and its unique combination of productivity and performance to enable fast, accurate, and agile cycle-level hardware modeling.

CHAPTER 4
PYDGINFF: ACCELERATING CYCLE-LEVEL MODELING USING META-TRACING JIT VMS

This chapter shows how meta-tracing JIT VMs can be used for accelerating cycle-level (CL) hardware modeling. I have built on the Pydgin framework described in Chapter 3 to complement fast-forward- and sampling-based methodologies to improve the performance of CL simulations. While existing techniques allow fast and accurate simulation, meta-tracing JITs can help also enable *agile* simulation. Agile simulation provides complete flexibility in terms of exploring the interaction between software and hardware. I introduce two techniques: JIT-assisted fast-forward embedding (JIT-FFE) to allow the meta-tracing JIT-based instruction-set simulator (ISS) to be *embedded* inside a more detailed CL simulator; and JIT-assisted fast-forward instrumentation (JIT-FFI) to quickly instrument the application to identify sampling points and keep parts of the architectural state warm.

4.1 Introduction

There is always a need for fast and accurate simulation of computer systems, but achieving these goals is particularly challenging when exploring new architectures for which native functional-first (trace-driven) simulation [pin15, CHE11, SK13, LCM+05, MKK+10, JCLJ08, MSB+05] is not applicable. Traditional approaches to improving the speed and accuracy of simulation for new architectures usually assume that the instruction-set architecture (ISA) and the entire software stack are fixed. For example, checkpoint-based sampling will collect a set of checkpoints from a profiling phase for a specific ISA and software stack, and then reuse these checkpoints across many different experiments. However, there is an emerging trend towards vertically integrated computer architecture research that involves simultaneously rethinking applications, algorithms, compilers, run-times, instruction sets, microarchitecture, and VLSI implementation. Vertically integrated computer architecture research demands *agile simulation*, which allows complete flexibility in terms of exploring the interaction between software and hardware. Agile simulation is certainly possible if one is willing to sacrifice either speed or accuracy. Table 4.1 illustrates how the state-of-the-art simulation methodologies can be fast and agile (e.g., instruction-set simulation with dynamic binary translation), accurate and agile (e.g., fast-forward-based sampling), or fast and accurate (e.g., checkpoint-based sampling). A newer trend is to use native execution to speed

	Fast	Accurate	Agile	Examples
DBT-Based ISS	●	○	●	[May87, MZ04, TJ07, LCL+11, BFT11, QDZ06, ABvK+11, LIB15]
FF-Based Sampling	○	●	●	[PHC03, WWFH03, WWFH06b, SPHC02, SAMC99, LS00]
CK-Based Sampling	●	●	○	[WWFH06a, VCE06, RPOM05]
NNFF-Based Sampling	●	●	◐	[SNC+15, Bed04, AFF+09, PACG11, SK13]
PydginFF	●	●	●	

Table 4.1: Simulator Methodologies – Comparison of different simulator methodologies for achieving fast, accurate, and agile simulation. DBT = dynamic-binary translation; ISS = instruction-set simulation; FF = fast-forward; CK = checkpoint; NNFF = native-on-native fast-forward acceleration.

up functional simulation when the target (i.e., the simulated architecture) and the host (i.e., the architecture running the simulator) are identical. Such "native-on-native" fast-forward acceleration is fast, accurate, and partially agile; this technique enables exploring changes to the software and microarchitecture, but does not enable research that involves changing the actual ISA. Our goal in this chapter is to explore a new approach that can potentially enable fast, accurate, and agile simulation for the entire computing stack.

Simple interpreter-based *instruction-set simulators* (ISSs) have significant overheads in fetching, decoding, and dispatching target instructions. Augmenting an ISS with *dynamic binary translation* (DBT) can enable fast and agile simulation. DBT identifies hot paths through the binary and dynamically translates target instructions on this hot path into host instructions. This eliminates much of the overhead of simple interpreter-based ISSs and enables simulation speeds on the order of hundreds of millions of instructions per second. While traditional DBT-based ISSs are known to be difficult to modify, recent work has explored automatically generating DBT-based ISSs from architectural description languages [ARB+05, QRM04, QM05, BFKR09, LIB15]. DBT-based ISSs are fast and agile, but these simulators do not accurately model any microarchitectural details.

Detailed simulation uses cycle- or register-transfer-level modeling to improve the accuracy of simulation, but at the expense of slower simulation speeds. Modern microarchitectural simulators run at tens of thousands of instructions per second, which means simulating complete real-world programs is practically infeasible (e.g., simulating a few minutes of wall-clock time can require months of simulation time). Researchers have proposed various *sampling techniques* to make detailed simulation of large programs feasible [WWFH03, CHM96, WWFH06b, SPHC02, SAMC99]. These techniques either use statistical sampling or light-weight functional profiling of the overall program to identify representative samples from the full execution. Detailed simulation is only required for the samples, yet these techniques can still maintain accuracy within a high confidence interval. Since the samples are usually a small ratio of the whole program, sampling can drastically reduce the amount of time spent in slow detailed simulation. However, the samples tend to be scattered throughout the entire execution, which raises a new challenge with respect to generating the correct architectural state (e.g., general purpose registers, page tables, physical memory) and potentially microarchitectural state (e.g., caches, branch predictors) to initiate detailed simulation of each sample.

Fast-forward-based (FF-based) sampling focuses on providing accurate and agile simulation. An interpreter-based ISS is used to "fast-forward" (FF) the program until the starting point of a sample, at which point the simulator copies the architectural state from the interpreter-based ISS into the detailed simulator [CHM96, SPHC02, SAMC99]. Some FF-based sampling schemes also require *functional warmup* where microarchitectural state is also generated during fast forwarding to ensure accurate detailed simulation of the sample [WWFH03, WWFH06b]. FF-based sampling significantly improves the simulation speed compared to detailed simulation without sampling, but it is still many orders-of-magnitude slower than DBT-based ISS. The execution time tends to be dominated by the interpreter-based ISS used during fast-forwarding, and as a consequence simulating a few minutes of wall-clock time can still require several days.

Native-on-native fast-forwarding-based (NNFF-based) sampling uses native execution instead of an interpreter-based ISS for fast forwarding. These techniques typically use virtualization to keep the host and simulated address spaces separate [SNC$^+$15, Bed04, AFF$^+$09, PACG11, SK13]. Because NNFF-based sampling uses much faster native execution for functional simulation, it can achieve fast, accurate, and partially agile simulation. NNFF-based sampling enables quickly changing the microarchitecture and software, but ISAs different than the host cannot run natively. New instructions and experimental ISAs cannot take advantage of native execution for fast-forwarding, making such studies unsuitable for NNFF-based sampling.

Checkpoint-based sampling focuses on providing fast and accurate simulation. An ISS is used to save checkpoints of the architectural state at the beginning of each sample [VCE06, WWFH06a, RPOM05]. Once these checkpoints are generated for a particular hardware/software interface and software stack, they can be loaded from disk to initiate detailed simulation of the samples while varying microarchitectural configuration parameters. Checkpoint-based sampling improves overall simulation time by replacing the slow FF step with a checkpoint load from disk. However, checkpoint-based sampling requires the hardware/software interface and software stack to be fixed since regenerating these checkpoints is time consuming. Because checkpoint generation is rare, the tools that profile and generate these checkpoints are usually quite slow; it can take many days to regenerate a set of checkpoints after changing the hardware/software interface or the software stack.

Section 4.2 provides background on DBT-based ISS and sampling-based simulation techniques. We make the key observation that while FF-based sampling is both accurate and agile, its speed suffers from slow FF. This motivates our interest in enabling fast, accurate, and agile simulation by augmenting FF-based sampling with recent research on DBT-based ISSs. However, there are several technical challenges involved in integrating these two techniques. DBT-based ISSs and detailed simulators use very different design patterns (e.g., page-based binary translation vs. object-oriented component modeling), data representations (e.g., low-level flat memory arrays vs. hierarchical memory modeling), and design goals (e.g., performance vs. extensibility). These differences significantly complicate exchanging architectural state between DBT-based ISSs and detailed simulators. Furthermore, instrumenting a DBT-based ISS to enable functional profiling and/or functional warmup can be quite difficult requiring intimate knowledge of the DBT internals.

In Sections 4.3 and 4.4, we propose *JIT-assisted fast-forward embedding* (JIT-FFE) and *JIT-assisted fast-forward instrumentation* (JIT-FFI) to enable fast, accurate, and agile simulation. JIT-FFE and -FFI leverage recent work on the RPython meta-tracing just-in-time compilation (JIT) framework for general-purpose dynamic programming languages [BCF+11a, BCFR09, pypa, Pet08, AACM07] and the Pydgin framework for productively generating very fast DBT-based ISSs [LIB15]. JIT-FFE enables embedding a full-featured DBT-based ISS into a detailed simulator, such that the DBT-based ISS can have *zero-copy* access (large data structures do not need to be copied) to the detailed simulator's architectural state. JIT-FFI enables productively instrumenting the DBT-based ISS with just a few lines of high-level RPython code, but results in very fast functional profiling and warmup. We have implemented JIT-FFE and -FFI in a new tool, called PydginFF, which can be integrated into any C/C++ detailed simulator.

Section 4.5 evaluates PydginFF within the context of the gem5 detailed simulator [BBB+11] and two different sampling techniques (periodic sampling through SMARTS [WWFH03, WWFH06b] and targeted sampling through SimPoint [SPHC02]) when running a variety of SPEC CINT2006 benchmarks. PydginFF is able to reduce the simulation time of FF-based sampling by over 10$\times$; simulations that previously took 1–14 days can now be completed in just a few hours.

To our knowledge, this is the first work to propose and demonstrate fast, accurate, and agile simulation through the creative integration of DBT-based ISSs and detailed simulation. Unlike related NNFF-based sampling approaches, our work allows the entire computing stack to be modified in an agile manner. We anticipate this approach would be particularly useful for studying ISA extensions or for exploring radical hardware acceleration techniques to improve the performance of emerging workloads where the software is not static (e.g., just-in-time compilation and optimization techniques). The primary contributions of this work are: (1) we propose JIT-assisted fast-forward embedding to elegantly enable zero-copy architectural state transfer between a DBT-based ISS and a detailed simulator; (2) we propose JIT-assisted fast-forward instrumentation to enable productive implementation of fast functional profiling and warmup; and (3) we evaluate these techniques within the context of PydginFF embedded into gem5 using SMARTS and SimPoint sampling techniques and show compelling performance improvements over traditional sampling with interpreter-based fast-forwarding.

4.2 Background

In this section, we provide brief background on DBT-based ISSs and sampling-based detailed simulation, including an overview of the SMARTS and SimPoint methodologies used in our evaluation.

4.2.1 DBT-Based Instruction Set Simulation

Instruction-set simulators (ISSs) facilitate software development for new architectures and the rapid exploration and evaluation of instruction-set extensions. In an interpreter-based ISS, a dispatch loop fetches and decodes target instructions before dispatching to a function that implements the instruction semantics. Dynamic-binary translation (DBT) can drastically improve the performance of interpreter-based ISSs by removing most of the dispatch-loop-based overheads. A DBT-based ISS still uses an interpreter for light-weight profiling to find frequently executed code regions. These hot regions are then translated into host instruction equivalents. The native assembly code generated using DBT is cached and executed natively whenever possible instead of using the interpreter. DBT-based ISSs require sophisticated software engineering since they include profiling, instruction-level optimizations, assembly code generation, code caching, and a run-time that can easily switch between interpreter- and DBT-based execution. Coordinating all these components while maintaining correctness and high performance makes DBT-based ISSs very hard to implement, maintain, and extend. However, promising recent work has demonstrated sophisticated frameworks that can automatically generate DBT-based ISSs from architecture description languages [PC11, WGFT13, QM03b].

At the same time, there has been significant interest in JIT-optimizing interpreters for dynamic programming languages. For example, JavaScript interpreters in web browsers make heavy use of JIT-optimizations to enable highly interactive web content [v8]. Another notable JIT-optimizing interpreter is PyPy for the Python language [BCF$^+$11a, BCFR09, pypa, Pet08, AACM07]. The PyPy project has created a unique development approach that utilizes the *RPython translation toolchain* to abstract the process of language interpreter design from low-level implementation details and performance optimizations. The interpreter developers write their interpreter (e.g., for the Python language) in a statically typed subset of Python called *RPython*. Using the RPython translation toolchain, an interpreter written in the RPython language is translated into C by going through *type inference*, *back-end optimization*, and *code generation* phases. The translated C code for the interpreter is compiled using a standard C compiler to generate a fast interpreter for the target language. In addition, the interpreter designers can add light-weight *JIT annotations* to the interpreter code (e.g., annotating the interpreter loop, annotating which variables in the interpreter denote the current position in the target program, annotating when the target language executes a backwards branch). Using these annotations, the RPython translation toolchain can automatically insert a JIT into the compiled interpreter binary. RPython separates the language interpreter design from the JIT and other low-level details by using the concept of a *meta-tracing JIT*. In a traditional tracing JIT, a trace of the target language program is JIT compiled and optimized. In a meta-tracing JIT, the trace is generated from the *interpreter* interpreting the target language program. Tracing JITs need to be specifically designed and optimized for each language, while meta-tracing JITs are automatically generated from the annotated language interpreter. The meta-tracing JIT approach removes the need to write a custom JIT compiler for every new language.

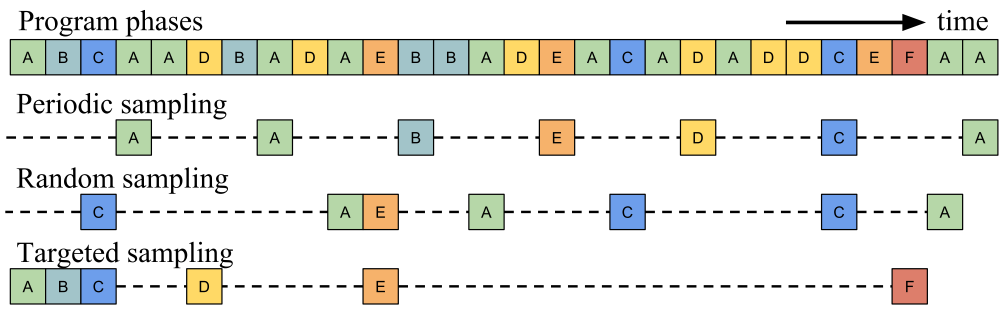


Figure 4.1: Sampling Methods – Different phases are represented with different letters. Sampling methods include periodic, random, and targeted. The selected representative samples are simulated using detailed simulation; the portion of the execution shown with a dashed line can use fast-forwarding or checkpointing.

Pydgin is a recent framework for productively building DBT-based ISSs [LIB15]. Pydgin makes use of the RPython translation toolchain to bridge the productivity-performance gap between interpreter- and DBT-based ISSs. An ISS in Pydgin is written as an interpreter in the RPython language, which allows it to be translated and compiled into an efficient binary. Pydgin also has the necessary annotations to allow RPython to automatically generate a very fast and optimized JIT. RPython's pseudocode-like syntax and the Pydgin library hide most performance-focused optimizations from the ISA definition, making modifying or adding new instructions very productive. Pydgin is a key enabler for the two techniques proposed in this chapter: JIT-assisted fast-forward embedding and JIT-assisted fast-forward instrumentation.

4.2.2 Sampling-Based Detailed Simulation

Figure 4.1 illustrates three approaches to sampling-based detailed simulation: random sampling, periodic sampling, and targeted sampling. Random sampling leverages the central limit theorem to enable calculating confidence bounds on performance estimates. Periodic sampling is an approximation of random sampling, and similar statistical tools can be used to ensure accuracy. Periodicity in the studied programs might skew the results for periodic sampling, however, this was shown not to be an issue for large benchmarks in practice [WWFH03, WWFH06b]. Targeted sampling requires a profiling step to find samples that are representative of different program regions. The profiling information can be microarchitecture-dependent [SAMC99] or microarchitecture-independent (e.g., based on basic-block structure [SPHC02] or loop/ call graphs [LPC06, LSC04]).

A key challenge in sampling-based detailed simulation is generating the architectural (and potentially microarchitectural) state to initiate the detailed simulation of each sample. Table 4.2

	Long-History Modeling	**Short-History Modeling**	**Collect Statistics**
Fast-Forwarding			
Functional Warmup	✓		
Functional Profiling			✓
Detailed Warmup	✓	✓	
Detailed Simulation	✓	✓	✓

Table 4.2: Simulation Terminology – Long-history modeling includes caches and branch predictors. Short-history modeling includes core pipeline. Collecting statistics might include profiling or microarchitectural statistics.

shows different types of simulation to produce the initial state for detailed simulation of each sample. *Fast-forwarding* is pure functional execution of the program and only the architectural state is modeled. This is the least detailed type of simulation, hence it tends to be the fastest. However, uninitialized microarchitectural state at the beginning of a sample can heavily bias the results. Researchers usually use some form of warmup to minimize this *cold-start bias*. The processor core pipeline contains relatively little "history" and thus requires warmup of a few thousand instructions. Caches, branch predictors, and transaction look-aside buffers contain much more "history" and thus require hundreds of thousands of instructions to minimize the cold-start bias [WWFH03, HS05, ELDJ05]. *Detailed warmup* will initiate detailed simulation before the start of the sample. Detailed warmup is good for warming up both long- and short-history microarchitecture, but is obviously quite slow. *Functional warmup* will update long-history microarchitectural state using a purely functional model during fast-forwarding. Because these long-history microarchitectural components tend to be highly regular structures, adding these models to fast-forwarding usually has a modest impact on simulation speed. A related type of simulation is *functional profiling*, which is used in some sampling methodologies to determine where to take samples. Similar to functional warmup, functional simulators can often use light-weight modifications to implement profiling with only a modest impact on simulation speed. Table 4.3 compares two common sampling-based simulation methodologies that we will use in our evaluation: SMARTS [WWFH03, WWFH06b, WWFH06a] and SimPoint [SPHC02, LPC06, LSC04, VCE06, PHC03].

SMARTS is one of the most well-known statistical sampling methodologies. This approach uses periodic sampling to approximate random sampling, which allows the authors to use statistical sampling theory to calculate confidence intervals for the performance estimates. While the original paper thoroughly evaluates different parameters such as the length of each sample, the number

	SMARTS	**SimPoint**
Sampling Type	periodic	targeted
Functional Profiling	optional	required
Num of Samples	10,000	maximum 30
Len of Samples	1000	10 million
Len of Detailed Warmup	2000	optional
Between Samples	functional warmup	fast forwarding

Table 4.3: Comparison of SMARTS and SimPoint – SMARTS has an optional profiling step to determine the length of the benchmark; SimPoint has a required functional profiling step to generate BBVs. Length of samples and detailed warmup are in instructions.

of samples, and the amount of detailed warmup, the paper ultimately prescribes for an 8-way processor: 10,000 samples, each of them 1000 instructions long, with 2000 instructions of detailed warmup [WWFH03]. SMARTS is able to use a relatively short length of detailed warmup by relying on functional warmup between samples. The authors determined that if functional warmup is unavailable and pure fast-forwarding is used instead, detailed warmup of more than 500,000 instructions (i.e., 500$\times$ the size of the sample) is required for some benchmarks.

SimPoint is one of the most well-known targeted sampling methodologies. SimPoint classifies regions of dynamic execution by their *signature* generated from the frequency of basic blocks executed in each region. SimPoint requires an initial functional profiling phase that generates basic block vectors (BBVs) for each simulation interval. Each element of the BBV indicates the number of dynamic instructions executed belonging to a particular basic block. Because the number of basic blocks is large, the dimensionality of the BBVs are reduced using random projection and then classified using the *k*-means clustering algorithm. One simulation interval from each cluster is picked to be a representative sample or *simulation point*. While the original SimPoint paper used 100 million instructions per sample with a maximum of 10 samples [SPHC02], a follow-up work used 1 million instructions per sample with a maximum of 300 samples [PHC03]. The most common parameters used in practice tend to be 10 million instructions per sample with a maximum of 30 samples. In contrast to SMARTS, SimPoint uses fewer but longer samples. This results in the inter-sample intervals of billions of instructions and thus SimPoint lends itself to pure fast-forwarding. Since the samples are very long, the effect of cold-start bias is mitigated and warmup is less critical.

4.3 JIT-Assisted Fast-Forward Embedding

One of the biggest obstacles in augmenting FF-based sampling with a DBT-based ISS is coordinating a single execution context for the target application between the ISS and the detailed simulator. The entire architectural state (and microarchitectural state in the case of functional warmup) needs to be communicated between these two simulators. These simulators often represent architectural state differently, using different data structures, in different alignments, and at different granularities. Performance-oriented DBT-based ISSs tend to represent the architectural state in forms that will facilitate high performance, but detailed simulators often choose more extensible approaches that allow running different ISAs and experiments. Another challenge is to ensure consistency of the architectural state for switching. Dirty lines in the modeled caches and uncommitted modifications to the architectural state (e.g., in host registers in DBT-based ISSs) need to be committed before switching. Once the entire architectural state is gathered in a consistent form, another challenge is marshalling this data, using an interprocess communication method such as writing/reading a file/pipe, and finally unmarshalling and reconstructing the data in the new simulator. Performing all of these tasks without incurring excessively long switching times can be quite challenging.

4.3.1 JIT-FFE Proposal

We propose JIT-assisted fast-forward embedding (JIT-FFE) to address this challenge. JIT-FFE enables the fast DBT-based ISS to be dynamically linked to the slow detailed simulator, obviating the need to run two different processes. Since both simulators share the same memory space, large data structures (e.g., the simulated memory for the target) can be directly manipulated by both the DBT-based ISS and the detailed simulator. This removes the need for marshalling, interprocess communication, and unmarshalling, and thus enables *zero-copy* architectural state transfer. JIT-FFE requires both simulators to obey the same conventions when accessing any shared data structures, even at the expense of slightly reduced performance for the DBT-based ISS. JIT-FFE does not require all data structures to be shared; for smaller architectural state (e.g., register file) simply copying the data between simulators is usually simpler. While JIT-FFE elegantly enables using a DBT-based ISS for fast-forwarding in SimPoint, JIT-FFE's ability to provide zero-copy state transfer is particularly effective in SMARTS due to large number of small samples (i.e., many switches between the DBT-based ISS and detailed simulator).

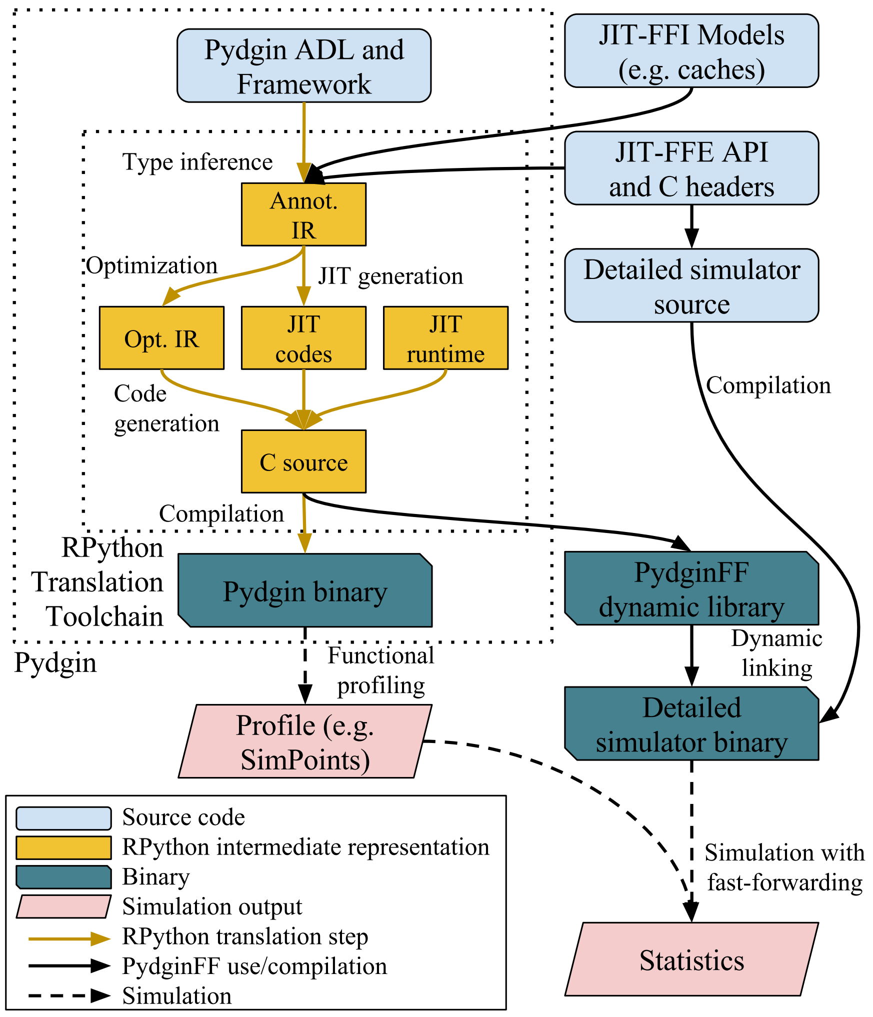


Figure 4.2: PydginFF Compilation and Simulation Flow

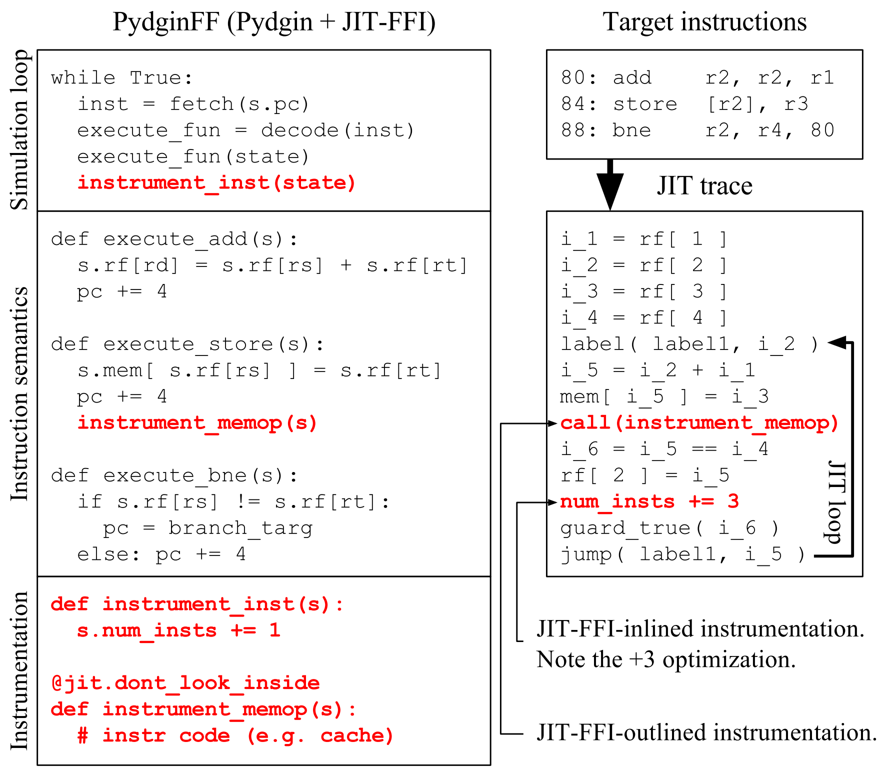
4.3.2 JIT-FFE Implementation

We have augmented Pydgin with JIT-FFE features, which we call PydginFF. PydginFF defines a C/C++ application-programming interface (API) that is visible to the detailed simulator. This API is used for getting and setting the architectural state, declaring a shared data structure for the simulated memory to enable zero-copy state transfer, and starting functional simulation. The API is defined both in the PydginFF source (in the RPython language) and in a C header file. RPython has a rich library to manipulate and use C-language types and structures, and declare C-language entry-point and call-back functions. Even though PydginFF is written in RPython, not C, the RPython translation toolchain translates PydginFF into pure C. This means that after the C/C++ detailed simulator dynamically links against PydginFF, the detailed simulator can directly interact with PydginFF without crossing any language boundaries. This is a key enabler that allows efficient zero-copy architectural state transfer between Pydgin ISS and the detailed simulator.

Figure 4.2 shows the compilation and simulation flow of PydginFF. Pydgin (without the PydginFF extensions) consists of an architectural description language (ADL) where the ISA is defined, and the framework which provides ISA-independent features and JIT annotations. The ADL and the framework go through the RPython translation toolchain which includes type inference, optimizations, code generation, and JIT generation. This produces the stand-alone JIT-optimized Pydgin binary that can be used for stand-alone functional profiling. The JIT-FFE extensions to Pydgin are also primarily in the RPython language, and go through the same RPython translation toolchain. In addition, we have configured the toolchain to also produce a dynamic library at the end of translation and compilation. C/C++ detailed simulators that are modified to take advantage of the PydginFF API can simply dynamically link against PydginFF and use the Pydgin JIT for fast-forwarding (and functional warmup) in sampled simulation. Targeted sampling methodologies such as SimPoint also need a functional profile of the application. For these, the stand-alone Pydgin binary can generate the functional profile which can be used by the detailed simulator to determine when to start sampling.

4.4 JIT-Assisted Fast-Forward Instrumentation

While JIT-FFE enables a DBT-based ISS to be used for fast-forwarding within a detailed simulator, common sampling methodologies also require instrumenting the functional simulation. For example, SMARTS requires the long-history microarchitectural state to be functionally modelled, and SimPoint uses basic-block vectors generated from functional profiling. Adding instrumentation to a traditional DBT-based ISS can be very challenging. DBT-based ISSs tend to be performance oriented and are not built with extensibility in mind. These simulators are usually written in low-level languages and styles in order to get the best performance. Moreover, the instrumentation code added to these simulators will not automatically benefit from the JIT and can significantly hurt performance.



PydginFF (Pydgin + JIT-FFI)
Target instructions
Simulation loop
Instruction semantics
Instrumentation
JIT trace
JIT loop
JIT-FFI-inlined instrumentation.
Note the +3 optimization.
JIT-FFI-outlined instrumentation.

```
while True:
  inst = fetch(s.pc)
  execute_fun = decode(inst)
  execute_fun(state)
  instrument_inst(state)

def execute_add(s):
  s.rf[rd] = s.rf[rs] + s.rf[rt]
  pc += 4

def execute_store(s):
  s.mem[ s.rf[rs] ] = s.rf[rt]
  pc += 4
  instrument_memop(s)

def execute_bne(s):
  if s.rf[rs] != s.rf[rt]:
    pc = branch_targ
  else: pc += 4

def instrument_inst(s):
  s.num_insts += 1

@jit.dont_look_inside
def instrument_memop(s):
  # instr code (e.g. cache)
```

```
80: add    r2, r2, r1
84: store  [r2], r3
88: bne    r2, r4, 80
```

```
i_1 = rf[ 1 ]
i_2 = rf[ 2 ]
i_3 = rf[ 3 ]
i_4 = rf[ 4 ]
label( label1, i_2 )
i_5 = i_2 + i_1
mem[ i_5 ] = i_3
call(instrument_memop)
i_6 = i_5 == i_4
rf[ 2 ] = i_5
num_insts += 3
guard_true( i_6 )
jump( label1, i_5 )
```

Figure 4.3: JIT-FFI Flow – JIT-FFI instrumentation (in **red**) can be added to Pydgin (in black). `instrument_inst` is called from the simulation loop to instrument every instruction; `instrument_memop` is called from the instruction semantics of the store instruction to instrument memory operations. The `@jit.dont_look_inside` decorator causes instrumentation to be JIT-FFI outlined. On the right is a simple loop in an example ISA and the resulting optimized JIT trace with instrumentation.

4.4.1 JIT-FFI Proposal

We propose JIT-assisted fast-forward instrumentation (JIT-FFI) to address this challenge. JIT-FFI allows the researcher to add instrumentation to the RPython interpreter, not to the JIT compiler itself. RPython's meta-tracing JIT approach generates a JIT compiler for the entire interpreter including instruction execution *and* instrumentation. This means instrumentation code is not just inlined but also dynamically JIT-optimized within the context of the target instruction stream. JIT-FFI inlining can produce very high performance for simple instrumentation. However, JIT-FFI inlining can reduce performance if the instrumentation includes complex data-dependent control flow, since this irregularity causes the meta-tracing JIT to frequently abort trace formation. JIT-FFI also includes support for JIT-FFI outlining where the instrumentation code is statically pre-compiled into an optimized function that can be directly called from the JIT trace. Figure 4.3 shows a

```
def instrument_memop( self, memop_type, address ):
  line_idx, tag  = self.get_idx_tag( address )

  # get if hit or not, and the way if it was a hit
  hit, way = self.get_hit_way( line_idx, tag )

  # If miss, get a victim way and update the tag
  if not hit:
    way = self.get_victim( line_idx )
    self.tag_array[ line_idx ][ way ] = tag

  # On write, set dirty bit
  if memop_type == WRITE:
    self.dirty_array[ line_idx ][ way ] = True

  # Update LRU bits
  self.update_lru( line_idx, way )

@jit.dont_look_inside
def get_hit_way( self, line_idx, tag ):
  for way in range( self.num_ways ):
    if tag == self.tag_array[ line_idx ][ way ]:
      return True, way
  return False, -1

def update_lru( self, line_idx, way ):
  self.lru_array[ line_idx ] = 0 if way else 1

def get_victim( self, line_idx ):
  return self.lru_array[ line_idx ]
```

Figure 4.4: JIT-FFI for Cache Warmup – RPython code to model a 2-way set-associative cache. The `@jit.dont_look_inside` decorator can be used to force JIT-FFI outlining.

simplified example of PydginFF code with JIT-FFI-inlined and -outlined instrumentation code, target instruction stream, and the resulting JIT trace.

JIT-FFI is critical to achieving fast and agile simulation using both SMARTS and SimPoint. For SMARTS, JIT-FFI enables very fast functional warmup of long-history microarchitectural state such as caches. For SimPoint, JIT-FFI enables very fast collection of basic-block vectors for determining representative samples.

4.4.2 JIT-FFI Implementation

We illustrate examples of instrumentation that users would write to take advantage of JIT-FFI in functional warmup and functional profiling, respectively. PydginFF provides hooks for users to add

```
# Called only when there are control-flow instructions
def instrument_ctrl( self, old_pc, new_pc, num_insts ):
  bb_idx = self.get_bb_idx( old_pc, new_pc )

  # bb_idx will never be equal to -1 in JIT trace
  if bb_idx == -1:
    # Register a new BB along with the size of the BB
    bb_idx = self.register_new_bb( old_pc, new_pc,
                          num_insts - self.last_num_insts )

  # Increment BBV entry by the size of the bb
  self.bbv[ bb_idx ].increment()
  self.last_num_insts = num_insts

# Get the index into the BBV table
@jit.elidable_promote()
def get_bb_idx( self, old_pc, new_pc ):

  # Construct BB signature, check if BB was seen before
  bb_sig = (old_pc << 32) | new_pc
  if bb_sig not in self.bbv_map:
    return -1

  return self.bbv_map[ bb_sig ]
```

Figure 4.5: JIT-FFI for BBV Generation – RPython code to generate BBV for SimPoint. `@jit.elidable_promote` is a JIT hint to enable aggressive JIT optimization.

custom instrumentation at different granularities (at instruction, memory operation, and control-flow operation levels with `instrument_inst`, `instrument_memop`, `instrument_ctrl` function calls respectively), and more of these hooks can be added. Figure 4.4 shows the JIT-FFI code within PydginFF to implement a functional model of a 2-way set-associative cache for use in SMARTS. The modeling is done in a rather straight-forward fashion, and the cache model maintains arrays for the tags (`tag_array`), dirty bits (`dirty_array`), and LRU bits (`lru_array`). The RPython language has a rich library of hints that can be given to the JIT, and an example for this can be seen in the `@jit.dont_look_inside` decorator, which leaves the hit lookup function as a call from the JIT trace and forces JIT-FFI outlining. Modeling the cache with a high-level language like RPython and then using simple JIT annotations make JIT-FFI both productive and fast. We evaluate a case study examining the effects of JIT-FFI inlining vs. outlining in set-associative and direct-mapped cache models in Section 4.5.4. Note that it is usually more efficient not to store data in the JIT-FFI caches, but simply retrieve the data from the main memory even on a cache hit (but still update tags and LRU bits accordingly). This means that the cache models do not benefit from JIT-FFE zero-copy architectural state transfer, and their states need to be copied between PydginFF and the detailed simulator.

Figure 4.5 shows the JIT-FFI code within PydginFF to implement basic-block vector (BBV) generation for use in SimPoint. Basic blocks in BBV generation are defined to be dynamic streams of instructions that have a control-flow instruction at the beginning and end, but not elsewhere. In the `instrument_ctrl` function, line 3 calls `get_bb_idx`, which returns a unique basic-block index or -1 if this basic block has not been seen before. The `@jit.elidable_promote` decorator on line 17 is a JIT annotation that constant-promotes the arguments (guaranteeing that the arguments will always be the same at the same point in the trace), and marks the function elidable (guaranteeing that this function does not have any side effects and always returns the same value). This annotation allows the function to be completely optimized away in the JIT trace and replaced with the corresponding constant basic-block index. In the JIT trace, the basic-block index will never be zero (it would have been observed before), so slower code to register a new basic block (lines 5–9) will be skipped. The only operation that will be inlined in the JIT trace is incrementing the corresponding BBV entry in line 12. This illustrates how JIT-FFI inlining can enable very high-performance instrumentation. We evaluate the effect of enabling JIT-FFI inlining versus outlining for BBV generation in Section 4.5.5.

4.5 Evaluation

We have implemented JIT-FFE and -FFI in a new tool, called PydginFF. Although PydginFF can be integrated into any C/C++ detailed simulator, we have chosen the popular gem5 detailed simulator [BBB$^+$11] for our evaluation, and we refer to the combined simulator as PydginFF+gem5.

4.5.1 Benchmarks

We quantify the performance of each simulator configuration (baseline SMARTS, baseline SimPoint, PydginFF+gem5 SMARTS, and PydginFF+gem5 SimPoint), using the SPEC CINT2006 benchmark suite. We use the ARMv5 instruction set for our target software, and a Newlib-based GCC 4.3.3 cross-compiler. We use the recommended optimization flags (`-O2`) for compiling the benchmarks. Three of the SPEC benchmarks, *400.perlbench*, *403.gcc*, and *483.xalancbmk* failed to

			SMARTS	SimPoint		
Benchmark	**Dataset**	**Dyn Inst**	**% Det Sim**	**Dyn Inst**	**# Spl**	**% Det Sim**
401.bzip2	`chicken 30`	195	0.020	194	25	0.200
429.mcf	`inp.in`	373	0.008	373	27	0.100
445.gobmk	`13x13.tst`	323	0.009	316	20	0.090
456.hmmer	`nph3 swiss41`	1112	0.003	952	13	0.020
458.sjeng	`ref.txt`	2974	0.001	2921	13	0.007
462.libquantum	`1397 8`	3069	0.001	3036	17	0.008
464.h264ref	`foreman_ref`	753	0.004	707	15	0.030
471.omnetpp	`omnetpp.ini`	1282	0.002	1254	3	0.004
473.astar	`BigLakes2048`	434	0.007	397	15	0.060

Table 4.4: Benchmarks – Dyn Inst = number of dynamic instructions when run to completion (in billions); % Det Sim = percentage simulated using the detailed simulator; # Spl = number of samples. SimPoint has slightly fewer dynamic instructions since the simulation can stop after the final sample.

	gem5 fun		gem5 det		Pydgin		gem5 SM		gem5 SP		PydginFF+gem5 SM			PydginFF+gem5 SP		
Benchmark	**IPS**	**T***	**IPS**	**T****	**IPS**	**T**	**IPS**	**T***	**IPS**	**T***	**IPS**	**T**	**vs. g5**	**IPS**	**T**	**vs. g5**
401.bzip2	2.4M	23h	54K	41d	613M	5.3m	2.2M	1.0d	2.0M	1.1d	44M	1.2h	20×	29M	1.9h	14×
429.mcf	2.4M	1.8d	60K	72d	487M	13m	2.0M	2.1d	1.9M	2.2d	34M	3.1h	17×	25M	4.2h	13×
445.gobmk	2.3M	1.6d	51K	74d	119M	45m	2.0M	1.8d	1.9M	1.9d	14M	6.3h	7×	40M	2.2h	20×
456.hmmer	2.3M	5.6d	67K	192d	582M	32m	2.1M	6.2d	1.9M	5.8d	49M	6.4h	24×	195M	1.4h	102×
458.sjeng	2.4M	14d	58K	596d	260M	3.2h	2.1M	16d	2.1M	16d	24M	1.5d	11×	160M	5.1h	76×
462.libquantum	2.6M	14d	66K	534d	605M	1.4h	2.2M	16d	2.1M	17d	93M	9.1h	43×	292M	2.9h	141×
464.h264ref	2.4M	3.6d	66K	133d	732M	17m	2.1M	4.1d	2.0M	4.0d	34M	6.2h	16×	157M	1.2h	77×
471.omnetpp	2.8M	5.4d	62K	240d	474M	45m	2.3M	6.4d	2.3M	6.3d	27M	13h	12×	209M	1.7h	90×
473.astar	2.5M	2.0d	64K	78d	386M	19m	2.1M	2.3d	2.0M	2.3d	31M	3.9h	15×	67M	1.6h	34×

Table 4.5: SMARTS and SimPoint Results – IPS = inst/second; T = simulation time (T* = extrapolated from 10B inst, T** = extrapolated from 500M inst); vs. g5 = speedup relative to gem5 baseline; fun = pure functional simulation; det = pure detailed simulation; SM = sampling-based simulation with SMARTS, SP = sampling-based simulation with SimPoint.

compile for the target configuration due to limited system call support in our cross-compiler so we omit these from our results. Table 4.4 shows the benchmark setup details. We used the *ref* datasets for all of the benchmarks, and picked one dataset in cases where there were multiple datasets. We also show the dynamic instruction counts of the benchmarks, which range from 200 billion to 3 trillion instructions.

4.5.2 Simulation Methodology

Our baseline simulation setup uses the gem5 [BBB+11] detailed microarchitecture simulator. We have implemented the SMARTS and SimPoint sampling techniques in gem5. Both baseline configurations use gem5's interpreter-based atomic simple processor for fast-forwarding, functional warmup, and functional profiling; and the cycle-level out-of-order processor model for detailed simulation. We use the ARMv5 instruction set for both PydginFF and gem5 with system-call emulation. For both configurations, the detailed microarchitecture models a 4-way superscalar pipeline with 4 integer ALUs, 2 AGUs, 32 entries in the issue queue, 96 entries in the ROB, and 32 entries each in load and store queues. We model a 2-way 32KB L1 instruction and a 2-way 64KB L1 data cache. The baseline SMARTS and SimPoint implementations use the configurations presented in Table 4.3. The SMARTS configuration only uses functional warmup between the samples, and SimPoint uses fast-forwarding until 500,000 instructions before the start of the simulation point and then switches to detailed warmup. Although optional, it is common practice to use detailed warmup with SimPoint.

As mentioned in Sections 4.3 and 4.4, we have used the open-source Pydgin DBT-based ISS [LIB15, pyd15] to develop PydginFF. We embed PydginFF into gem5 (PydginFF+gem5) and evaluate the performance, again using SMARTS and SimPoint, against the baseline SMARTS and SimPoint implementations with gem5 alone. The PydginFF+gem5 configurations use the DBT-based ISS for fast-forwarding, functional warmup, and functional profiling, and the same gem5 cycle-level microarchitectural model for detailed warmup and detailed simulation. We use PyPy/RPython 2.5.1 to translate PydginFF, GCC 4.4.7 to compile both PydginFF and gem5, and SimPoint 3.2 to generate simulation points from BBVs. We run all experiments on an unloaded 4-core 2.40 GHz Intel Xeon E5620 machine with 48 GB of RAM.

Table 4.4 shows sampling-related statistics of the benchmarks we used. A difference between the two sampling techniques is that SimPoint requires running the target program until the end of the last simulation point instead of running it to completion. However, it can be seen that the last simulation points tend to be close to the completion of the program, so this benefit is minimal. The number of simulation points in SimPoint can be varied as well, and our benchmarks showed a large range from 3 to 27. This number determines how much total detailed simulation needs to take place, which can affect the simulation performance. Since SMARTS uses the same number of samples for each benchmark, the total detailed simulation is the same. The table also shows the percentage

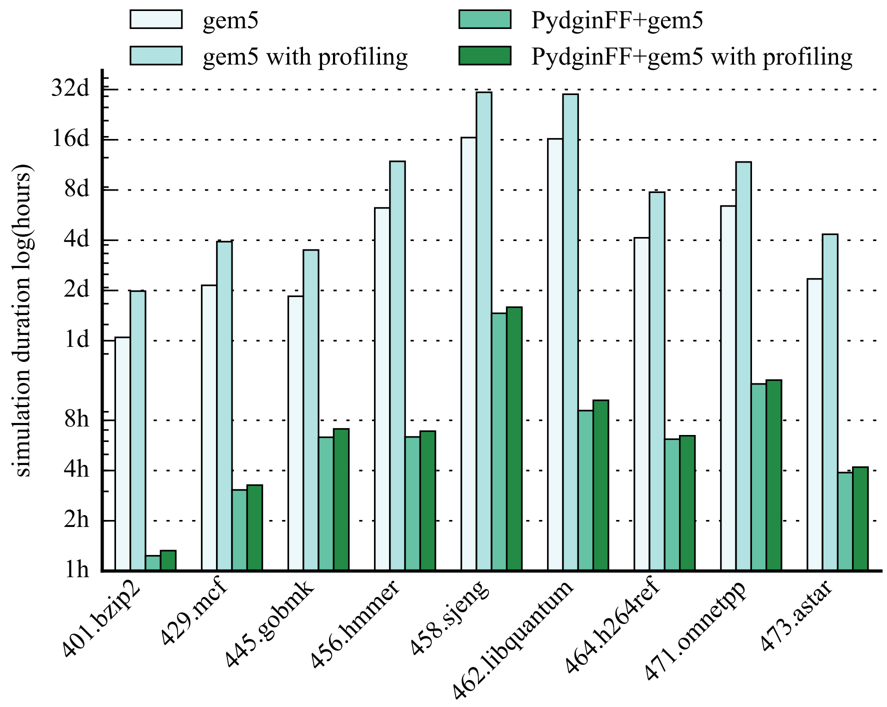


Figure 4.6: SMARTS Results – gem5 and PydginFF+gem5 results using SMARTS. Note the log scale for duration. with profiling = functional profiling added on top of simulation time.

of detailed simulation (including detailed warmup) that takes place in each sampling technique. These values are extremely low, indicating that fast-forwarding does indeed constitute 99+% of the simulation and motivating the need for faster fast-forwarding and functional warmup.

4.5.3 Overall Results

Table 4.5 shows the instructions per second (IPS) and elapsed time results of pure gem5, pure Pydgin, and PydginFF+gem5 configurations. gem5 simulator performance is very consistent across different benchmarks: around 2.5 MIPS and 60 KIPS for functional and detailed simulations, respectively. Even on the functional simulator, the simulations can take many days. Simulating the entire lengths of the longer benchmarks in the detailed model would take well over a year, which clearly is not feasible. Note that the table shows the results for gem5 functional simulation without any cache modeling. However, we also ran gem5 functional simulation with cache modeling (e.g., for functional warmup) and the results were very close to without caches. The Pydgin column shows the performance that a pure DBT-based ISS can achieve, between 100–700 MIPS. It should be noted that DBT-based ISS performance is more varied compared to interpreter-based ISSs like

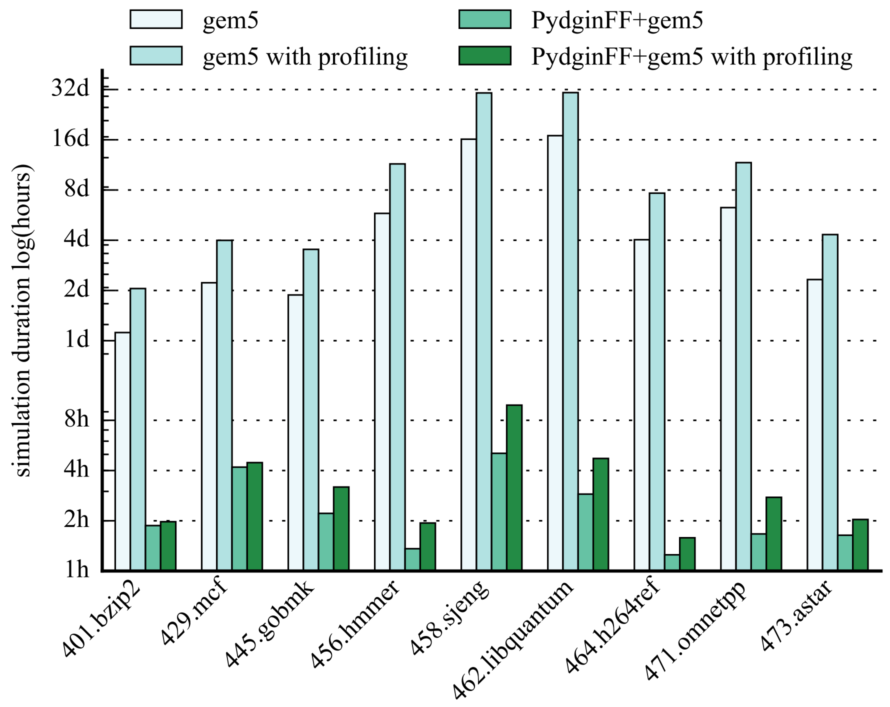


Figure 4.7: SimPoint Results – gem5 and PydginFF+gem5 results using SimPoint. Note the log scale for duration. with profiling = functional profiling added on top of simulation time.

the gem5 atomic model. This variation is due to dynamic optimizations done on instruction streams: some instruction streams benefit more than others. The longest-running benchmark took only about 3 hours on the DBT-based ISS.

The SMARTS results can be seen in Table 4.5 (*gem5 SM* and *PydginFF+gem5 SM* columns) and in Figure 4.6. The plot shows the total duration of simulation in log timescale. In the baseline gem5 SMARTS configuration, simulations can take 1–16 days, similar to gem5 functional model. This is because the vast majority of instructions are being executed in the functional warmup mode. To guarantee low error bounds, SMARTS needs a sufficient number of samples. A functional profiling step hence might be necessary to determine the total number of instructions, which can then be used to compute the distance between samples to reach the target number of samples. Adding this optional functional profiling phase roughly doubles the simulation time on gem5. PydginFF+gem5 using SMARTS sampling performs much better: between one hour to less than two days. The speedup of this technique compared to gem5 can be seen in Table 4.5, which is well over an order of magnitude for most of the benchmarks.

The SimPoint results are also shown in Table 4.5 (*gem5 SP* and *PydginFF+gem5 SP* columns) and in Figure 4.7. The baseline gem5 SimPoint results are similar to SMARTS because the

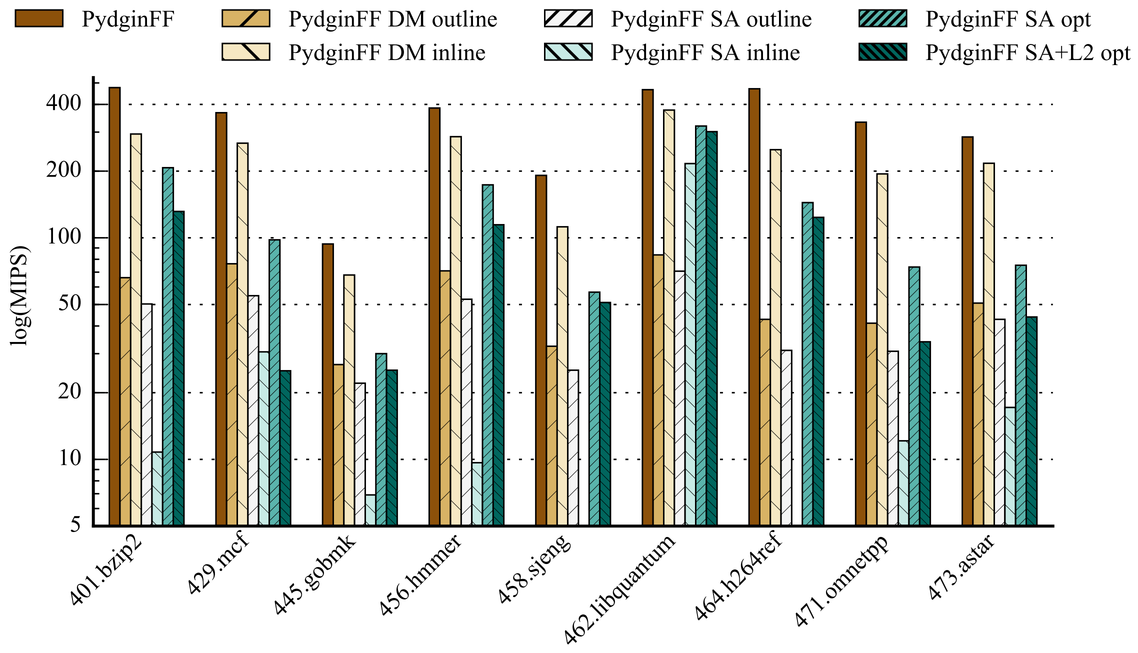


Figure 4.8: PydginFF Cache Modeling – PydginFF = PydginFF with no cache modeling; PydginFF DM/SA outline/inline = PydginFF modeling a direct-mapped/set-associative cache modeling with JIT-FFI outlining/inlining; SA opt = set-associative cache with optimized JIT-FFI inlining; SA+L2 opt = L1 and L2 caches (both set-associative) with optimized JIT-FFI inlining. Note that this study uses stand-alone PydginFF with no gem5 detailed simulation. 458.sjeng and 464.h264ref with a JIT-FFI inlined SA cache model were aborted due to using too much memory.

execution time again is dominated by the highly predictable performance of the gem5 functional model. PydginFF+gem5 manages to get even better speedups compared to SMARTS (in the range 13–141×) with a maximum execution time of only about five hours. Even including functional profiling, which SimPoint requires for every new binary, PydginFF+gem5 only takes about nine hours in the worst case, compared to 32 days using the baseline. The reason why SimPoint performance is better on PydginFF+gem5 compared to SMARTS is because SimPoint uses fast-forwarding instead of functional warmup, and fast-forwarding on our DBT-based ISS is much faster, as will be shown in Section 4.5.4. One final thing to note is that longer-running benchmarks (e.g., 462.libquantum and 458.sjeng) can get much better speedups compared to shorter-running benchmarks (e.g., 401.bzip2 and 429.mcf) due to the lower ratio of detailed simulation to functional simulation.

4.5.4 JIT-FFI Case Study: Cache Modeling

The SMARTS technique requires functional warmup to be used instead of fast-forwarding to generate the microarchitectural state before the detailed simulation starts. Functional warmup requires the long-history microarchitectural structures (e.g., caches) to be simulated in the functional model. We used JIT-FFI to implement direct-mapped and set-associative caches in PydginFF. In Figure 4.8, we compare the performances of stand-alone PydginFF (without gem5 detailed simulation) using direct-mapped and set-associative cache modeling to PydginFF without caches. We also compare the effects of using JIT-FFI inlining and outlining when implementing the cache model. In this study, PydginFF without caches models virtual memory, so its performance is slightly lower than unmodified Pydgin that uses raw memory. A JIT-FFI outlined direct-mapped cache significantly reduces performance compared to PydginFF without caches (up to 10×). However, JIT-FFI inlining can bring the performance back to reasonable levels (within 50% of PydginFF without caches). Outlined set-associative cache unsurprisingly has an even worse performance than outlined direct-mapped cache due to the increased complexity in the cache model. However, unlike the direct-mapped cache where JIT-FFI inlining helps, JIT-FFI-inlined set-associative cache performs worse that outlining.

The reason for this slowdown is because the set-associative cache model has data-dependent control-flow at line 22 in Figure 4.4. This if statement checks each cache way for the tag, and this information is recorded in the generated trace. Any time the same static instruction loads from or stores to a memory address that belongs to another cache way, the trace is aborted, which causes the observed speed degradation. These frequent trace aborts usually cause new traces to be generated, each of them with a different control-flow path. For certain benchmarks, the explosion of compiled traces can use up all of the host memory, as was observed in 458.sjeng and 464.h264ref. However, in one benchmark, 462.libquantum, the static loads and store instructions usually hit the same way, and do not display this pathological behavior. Due to these observations, PydginFF uses JIT-FFI outlining for set-associative cache models including for the results in the previous section.

We have also implemented a more optimized set-associative cache model using JIT-FFI inlining (shown as PydginFF SA opt in Figure 4.8). This uses a relatively large (~1 GB) lookup table for the entire address space to map every cache line to its corresponding way in the cache, or an invalid value to indicate a miss. This removes the data-dependent control flow, and manages to increase the performance to 50–310 MIPS for most benchmarks. Note that for a 64-bit ISA, this optimized

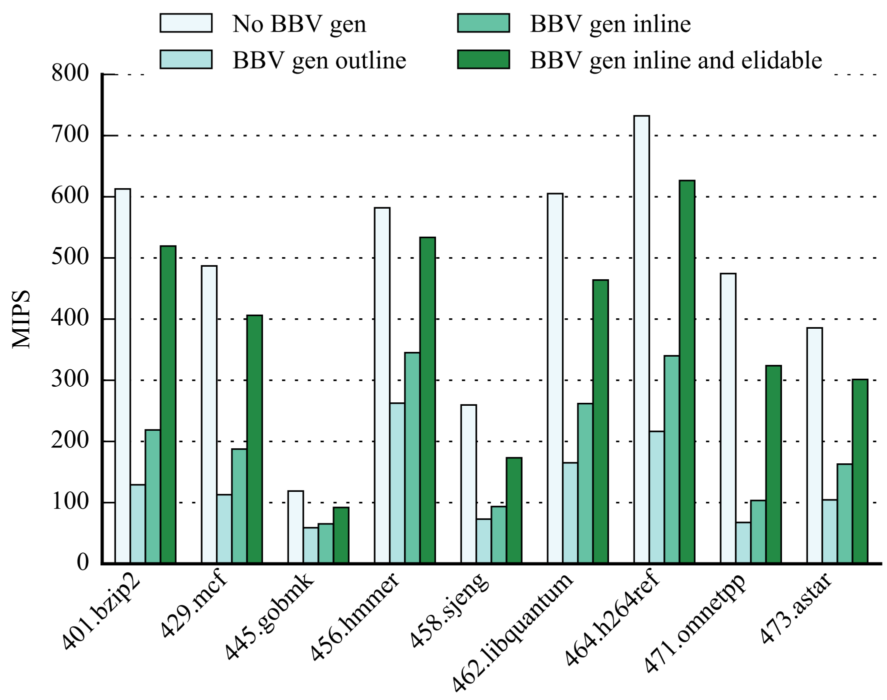


Figure 4.9: BBV Generation with Optimizations – No BBV gen = unmodified Pydgin that does not generate BBVs; BBV gen outline/inline = BBV generation with JIT-FFI outlining/inlining; elidable = uses the `@jit.elidable_promote` hint. Note that this study uses stand-alone Pydgin and PydginFF with no gem5 detailed simulation.

implementation might require a prohibitively large lookup table. Figure 4.8 also shows the impact of adding a set-associative L2 cache; PydginFF is still able to achieve simulation performance between 30–300 MIPS for most benchmarks.

4.5.5 JIT-FFI Case Study: BBV Generation

Basic-block vectors (BBVs) generated from the functional profiling step are crucial for the SimPoint sampling methodology. Because these BBVs need to be generated any time the target software changes, it is important that the BBV generation is fast and does not prevent agile simulation. We used JIT-FFI to implement the BBV generation as explained in Section 4.4.2. We compared unmodified Pydgin to BBV generation with JIT-FFI outlining, JIT-FFI inlining, and using the more advanced `@jit.elidable_promote` hint. These experiments were done on stand-alone Pydgin and PydginFF without gem5 detailed simulation. Figure 4.9 shows the results for this case study. The goal with inlining and eliding optimizations is to approach the unmodified Pydgin performance, hence have the least possible overhead in generating the BBVs. Figure 4.9 shows that unoptimized (outlined) BBV generation can have more than $7\times$ slowdown compared to unmodified Pydgin. With the addition of simple JIT annotations, the BBV generation overhead shrinks to around 20%.

4.6 Related Work

There has been significant prior work on DBT-based instruction-set simulators. Seminal work on DBT techniques provided significant performance benefits over traditional interpretive simulation [May87, CK94, WR96, MZ04]. These performance benefits have been further enhanced by optimizations which reduce overheads and improve code generation quality of just-in-time compilation (JIT) [TJ07, JT09, LCL+11]. Current state-of-the-art ISSs incorporate parallel JIT-compilation task farms [BFT11], multicore simulation with JIT-compilation [QDZ06, ABvK+11], or both [KBF+12]. This work builds on top of Pydgin [LIB15], which uses a slightly different approach for building DBT-based simulators. Instead of a custom JIT, Pydgin leverages the RPython framework [BCF+11a, BCFR09, pypa, Pet08, AACM07], which allows encoding the architecture description in pure Python. The RPython framework then adds a JIT to the interpreter and translates it to C. There has also been prior work on embedding the cycle-approximate hardware model directly in the JIT [Fra08, BFT10]. While these approaches tend to be fast and agile for changing the software, they are not agile for changing the modeled microarchitecture.

To reduce simulation time, KleinOsowski et. al proposed manually crafting smaller datasets for the SPEC 2000 suite that were representative of the larger datasets [KFML00]. However, manually manipulating the datasets is not agile for frequent changes in the software stack. Sampling approaches, in contrast, are automated, so they are much more agile in the face of changing software. Researchers proposed picking samples randomly [LPP88, CHM96], periodically [WWFH03, WWFH06b], and targeted [SPHC02, SAMC99, LS00]. Random and periodic sampling methodologies can be used to prove error bounds. Targeted sampling uses either microarchitectural [SAMC99] or microarchitecture-independent [SPHC02, LS00] metrics, to identify samples that are representative of the overall execution. A notable work on SimPoint in the context of modeling different ISAs is Perelman et. al, where the authors propose techniques to synchronize the simulation points on different ISAs to correspond to the same program phases [PLP+07]. This work is orthogonal to PydginFF.

Checkpointing is a very popular technique to reduce the simulation time further. Architectural state is usually very large, so checkpoint sizes on the disk is a concern, especially when there are many checkpoints [VCE06]. However, there are proposals to reduce the checkpoint size by analyzing which memory addresses are actually used in each sample, and then only storing these values in the checkpoint [WWFH06a, VCE06], and over time cheaper disk space made this issue less of a concern. The most serious shortcoming of checkpointing is its lack of agility: every time the software or the program inputs change, checkpoints need to be re-generated.

Other techniques to improve the simulation time includes Perelman et. al, which proposes an improved version of the SimPoint algorithm where simulation points that occur early in the program execution are prioritized to reduce the amount of fast-forwarding necessary [PHC03]. This technique can be used in conjunction with our proposal to improve simulation time. Wisconsin Wind Tunnel is one of the earliest uses of direct execution to speed up simulator performance [RHL+93, MRF+00]. Patil et. al use the Pin [LCM+05,pin15] dynamic binary instrumentation tool to generate the profiling information to be used by SimPoint [PCC+04]. Similarly, Szwed et. al propose native execution for fast-forwarding, by re-compiling the original code to explicitly extract the architectural state to be copied to a detailed simulator for sampling [SMB+04]. However, both of these approaches only work if the target ISA is the same as the host ISA, which is usually not the case for new ISA research or studying new ISA extensions. Schnarr and Larus proposed using native execution coupled with microarchitectural modeling of an out-of-order processor [SL98]. They use memoization to cache timing outcomes of previously seen basic blocks and starting microarchitectural states and skip detailed simulation if they hit in the timing cache. Brankovic et. al looked into the problem of simulating hardware/software co-designed processors which usually include a software optimization layer [BSGG14]. For accurate simulation, the software optimization layer needs to be warmed up in addition to the microarchitecture. The authors detect ways to determine when the optimization layer is warmed up, and PydginFF would be an orthogonal technique to speed up the software-optimization-layer warmup.

Another notable work that makes sampling-based simulation fast is Full Speed Ahead (FSA) [SNC+15], where the authors also acknowledge the need for agility in the context of hardware/-software co-design. FSA uses virtualized native execution for fast-forwarding until the samples and also uses zero-copy memory transfer when switching between the virtualized native execution and detailed model. However, because FSA relies on native execution, it is not suitable when the ISA is different than the host in the context of ISA extensions or modeling new ISAs. Furthermore, fast functional warmup is not supported on FSA, which is required by SMARTS. The COTSon [AFF$^+$09] project uses fast-forward-based sampling, where the SimNow ISS [Bed04] is used to fast-forward between the samples. SimNow uses DBT techniques to speed up the simulation. However, SimNow is not open-source and only supports the x86 ISA, so it is not possible to use this flow for agile ISA extension developments. Other recent attempts to make simulation fast (e.g., Graphite [MKK$^+$10], CMPSim [JCLJ08], ZSim [SK13], MARSS [PACG11], and Sniper [CHE11]) also rely on dynamic binary instrumentation, so are not suitable when the target ISA is different than the host. The only other simulator that uses JIT technology for fast-forwarding that we are aware of is ESESC [AR13]. ESESC uses JIT-optimized QEMU ISS for fast-forwarding and functional warmup. However, QEMU is a performance-focused ISS that sacrifices productivity for performance, which makes experimenting with ISA extensions challenging. Pydgin focuses both on productivity and performance, and gives researchers agility in modifying the entire computation stack.

4.7 Conclusion

State-of-the-art simulation methodologies can be fast and agile (e.g., instruction-set simulation with dynamic binary translation), accurate and agile (e.g., fast-forward-based sampling), or fast and accurate (e.g., checkpoint-based sampling). Native-on-native fast-forward-based sampling is fast, accurate, but partially agile. In this chapter, we have proposed JIT-assisted fast-forward embedding (JIT-FFE) and JIT-assisted fast-forward instrumentation (JIT-FFI) that elegantly enable augmenting a detailed simulator with a DBT-based ISS. We have implemented JIT-FFE and -FFI in a new tool, called PydginFF, and we have evaluated our approach within the context of the gem5 detailed simulator and two different sampling techniques (periodic sampling with SMARTS and targeted sampling with SimPoint). Our results show that PydginFF is able to reduce the simulation time of fast-forward-based sampling by over $10\times$, truly enabling fast, accurate, and agile simulation.

PydginFF opens up a number of interesting directions for future research. JIT-FFE's ability to provide zero-copy state transfer and JIT-FFI's ability to easily model microarchitectural components can be applied to branch predictors and TLBs. PydginFF can be integrated with any C/C++ microarchitectural simulator, so we are exploring integration with simulators that support a functional/timing split or even register-transfer-level models. Finally, PydginFF enables new kinds of vertically integrated research. For example, PydginFF can enable exploring hardware acceleration and ISA specialization for emerging workloads such as JIT-optimized interpreters of dynamic programming languages, using fast, accurate, and agile simulation.

CHAPTER 5
MAMBA: ACCELERATING RTL MODELING USING META-TRACING JIT VMS

This chapter explores the use of meta-tracing JIT VMs to accelerate register-transfer-level (RTL) hardware simulation methodologies. There has been recent interest in hardware design using domain-specific languages (DSL) embedded in modern high-level languages such as Python. Using an embedded DSL allows taking advantage of the productivity features of the host languages for advanced parameterization, testing, and mixed-level modeling [LZB14]. While productive, this approach can suffer from poor performance, even if using JIT VMs. To highlight how meta-tracing JITs can address performance issues, we introduce Mamba, a JIT-aware hardware generation and simulation framework (HGSF), and HGSF-aware JIT techniques. Combined, these techniques allow us to reach the performance of commercial Verilog simulators.

Mamba has been created in collaboration with Shunning Jiang [JIB18], and I have co-led the JIT optimizations to make Mamba high-performance. In addition, this chapter has been included to highlight that meta-tracing JIT technology can be promising for RTL simulation in addition to FL and CL models.

5.1 Introduction

With the rising importance of domain-specific hardware and increasing complexity of hardware design with the latest process technologies, there has been an influx of interest in using high-level general-purpose programming languages for hardware design. To complement, and sometimes completely replace, lower level register-transfer level (RTL) hardware description languages (HDLs) (e.g., Verilog and VHDL), hardware generation frameworks (HGFs) have been gaining traction. HGFs make use of general-purpose language features to improve the productivity in hardware design and reduce boilerplate and generate HDL descriptions of the hardware to be used by the traditional simulation or synthesis tools.

There has been a wide array of approaches in implementing HGFs starting with *hardware preprocessing frameworks*, where the high-level language is used as a preprocessor language alongside the HDL (e.g., Scheme mixed with Verilog in Verischemelog [JB99], Perl mixed with Verilog in Genesis2 [SAW^{+}10]), and *hardware generation-only frameworks*, where the parameterization, static elaboration, test bench generations, and behavioral modeling are unified in a high-level host language (e.g., Haskell in Lava [BCSS98], standard ML in HML [LL00], Scala in Chisel [BVR+12], Python in Stratus [BDM+07], PHDL [Mas07]). While these frameworks are promising, they lack infrastructure to simulate hardware directly, and rely on converting the hardware description to HDL and use HDL simulation tools. This has the drawback of potentially long modify-translate-simulate development cycle, and a semantic gap between the simulation and the hardware description. More recently, the addition of simulation capability directly at the high-level language to HGFs give rise to *hardware generation and simulation frameworks* (HGSFs) manage to address these pitfalls (e.g., Java in JHDL [BH98], Haskell in CλaSH [BKK+10], Python in MyHDL [Dec04], PyRTL [CTD+17], Migen [mig], PyHDL [HMLT03]). In particular, PyMTL showed the promise of productively using a Python-based embedded DSL for hardware generation and simulation for functional-, cycle-, and register-transfer-level modeling [LZB14].

While it can yield productivity benefits, hardware simulation directly in the host high-level language tends to be slow. This is even more pronounced when the host language is a dynamic language such as Python. While PyMTL took advantage of the general-purpose meta-tracing JIT of PyPy, that approach still resulted in more than two orders of magnitude slowdown compared to fast Verilog simulation [LZB14]. In this chapter, we explore additional meta-tracing JIT techniques to address this performance gap. In particular: (1) we describe five JIT-aware HGSF techniques (static scheduling, schedule unrolling, heuristic topological sort, trace breaking, and block consolidation); (2) we describe two HGSF-aware JIT techniques (support for performance-critical data types and huge loops); and (3) we introduce Mamba which uses these techniques to match the performance of commercial HDL simulators while maintaining the productivity benefits of using a single host language for parameterization, static elaboration, test bench generation, behavioral modeling, and simulation.

5.2 The Mamba HGSF

Like earlier versions of PyMTL, Mamba supports behavioral modeling using concurrent structural composition, positive-edge-triggered update blocks, and combinational update blocks. Mamba's syntax is similar in spirit to PyMTL. Figure 5.1 shows an example component with combinational update blocks that use the `@s.update` decorator. Each of the combinational update

```python
class ExampleComponent( RTLComponent ):

  def __init__( s ):
    s.a = Reg( Bits1 )
    s.b = Reg( Bits1 )
    s.c = Reg( Bits1 )
    s.d = Reg( Bits1 )

    @s.update
    def update_a():
      if   s.b.out: s.a.in_ = ... # A'
      else:         s.a.in_ = ... # A

    @s.update
    def update_b():
      if   s.a.out: s.b.in_ = ... # B'
      elif s.c.out: s.b.in_ = ... # B''
      else:         s.b.in_ = ... # B

    @s.update
    def update_c():
      s.c.in_ = ...

    @s.update
    def update_d():
      s.d.in_ = ...

```

Figure 5.1: Example PyMTL/Mamba Component – Components consist of update blocks that use the `@s.update` decorator. Mamba finds a valid static schedule that respects the dependencies between the components. The update blocks in this particular example do not have any dependencies between them, so any schedule is valid.

blocks define the signal value to the inputs to four 1-bit registers: `a`, `b`, `c`, and `d`. Mamba also leverages Python's built-in reflection features, particularly abstract-syntax-tree (AST) self-parsing, to determine which variables are read or written in an update block. Sensitivity information is constructed based on readers/writers of the same variable in different blocks.

Table 5.1 includes results for an iterative divider and a simple single- and multi-core RISC-V design described in more detail in Section 5.3. In the rest of this section, we describe each technique in more detail and discuss various implementation trade-offs involved with that technique.

5.2.1 JIT-Aware HGSF

As a starting point, we implemented event-driven simulation in Mamba using a very similar technique to PyMTL. Figure 5.2 shows the schedule choices that affect the performance. Like

```
def sim_loop():
  while not model.done():
    model.tick()
    ncycles += 1

def tick_event():
  q = set([update_a, update_b, update_c, update_d])
  while q:
    update = q.pop()
    update()
    # Add back all combinational update blocks to q whose sensitivity
    # lists are triggered after calling update().
    q.update( get_sensitivity_triggered_blocks() )

def tick_static():
  # Statically schedule A->B->C->D that respects dependencies.
  for update in [update_a, update_b, update_c, update_d]:
    update()

def tick_static_unroll():
  # Unroll static schedule A->B->C->D.
  update_a()
  update_b()
  update_c()
  update_d()

def tick_static_unroll_toposort():
  # Prioritize non-branchy combinational update blocks first C->D->A->B.
  update_c()
  update_d()
  update_a()
  update_b()

def tick_static_unroll_toposort_break():
  # Create meta-update blocks that limit branchiness C->D->A.
  def metablock_1():
    update_c()
    update_d()
    update_a()
  # Call the combinational update (and meta-update) blocks ( C->D->A )->B.
  metablock_1()
  # Use advanced application-level JIT hint to stop tracing between
  # meta-update blocks.
  pypyjit.dont_trace_here( ... )
  update_b()
```

Figure 5.2: Mamba Combinational Update Block Schedules – `sim_loop` is the simulation loop that iterates every cycle and `model.tick()` calls one of the functions with the corresponding combinational update block scheduling strategies for the example in Figure 5.1: `tick_event` = event-driven scheduling, `tick_static` = static scheduling, `tick_static_unroll` = unrolled static scheduling, `tick_static_unroll_toposort` = unrolled static scheduling with heuristic topological sort, `tick_static_unroll_toposort_break` = unrolled static scheduling with heuristic topological sort and trace breaking.

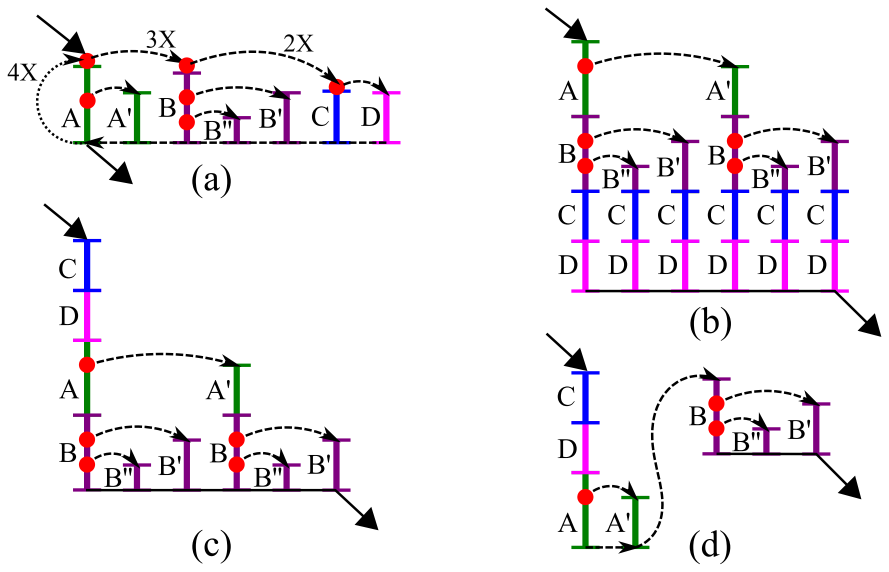


Figure 5.3: Meta-Traces of One Simulated Cycle – (a) Event driven and static scheduling; (b) schedule unrolling; (c) heuristic topological sort; (d) trace breaking. A,B,C,D = traces of update blocks; red dots = guards that have bridges compiled from (connected by dashed arrows); A',B',B" = conditional paths in update blocks that result in bridges; 4X,3X,2X = how many times the jump occurs in a simulated cycle; solid arrows = entry from and exit to the cycle loop.

Technique	Divider	1-Core	32-core
Event-Driven	24K CPS	6.6K CPS	65 CPS
JIT-Aware HGSF			
+ Static Scheduling	13×	2.6×	1.1×
+ Schedule Unrolling	16×	24×	0.2×
+ Heuristic Toposort	18×	26×	0.3×
+ Trace Breaking	19×	34×	1.5×
+ Consolidation	27×	34×	42×
HGSF-Aware JIT			
+ RPython Constructs	96×	48×	61×
+ Support Huge Loops	96×	49×	67×

Table 5.1: Mamba Performance – The baseline is event-driven simulation in Mamba. Each row adds a new technique upon all previous ones. All results are with PyPy. CPS = simulated cycles per second.

PyMTL, we use two nested loops: an outer loop for simulated cycles (lines 1–4), and an inner loop over an event queue of combinational update blocks. The `tick_event` function (lines 6–13) shows a simplified implementation of the event-driven schedule. The event queue, `q`, is implemented using a Python set data structure. The schedule loop iterates until `q` is empty. In each iteration, one item is removed from `q` and its combinational update block called. The framework also keeps track of combinational dependency sensitivity lists for each combinational update block. If any of these values that are in the sensitivity lists change, then the corresponding update blocks would be inserted back into `q` at the end of the iteration.

Table 5.1 shows the performance of event-driven Mamba simulation executing on a tracing-JIT VM. Because each iteration of the inner loop is a different update block, the tracing JIT compiles a different trace for each of these update blocks. The tracing JIT will then insert a guard at the beginning of each trace to check if that trace is compiled for the called update block. Failing that guard would cause a jump to the next bridge and check the guard there. Figure 5.3(a) illustrates this scenario using a cartoon representation of traces, guards, and bridges. Each trace with a different color and letter represents a trace from a different update block. The letters A, B, C, and D correspond to the update blocks `update_a`, `update_b` etc. in Figure 5.1. The letters A', B', and B" correspond to different control flow paths within the update blocks (lines 11, 16, and 17 respectively). Unfortunately, in a tracing JIT, these guards create a pathological chain of bridges for the inner loop. Executing the n-th compiled update block will result in failing the first $n-1$ guards. In other words, the number of guard failures in an entire simulated cycle scales *quadratically* with the total number of update blocks, which becomes a scaling bottleneck. PyPy currently does not have a mechanism to more efficiently do function dispatching (e.g., using switch/case). Furthermore, with smaller traces, the escape analysis in the compiler would not be able to find as many object dereferences and allocations that can be optimized away. Finally, enqueuing dependent blocks only when a signal's value changes requires an extra data-dependent check after every assignment. So while event-driven simulation can be efficient when most signals are stable, it can also create a number of challenges for tracing JITs. The JIT-aware HGSF techniques described in this section help mitigate many of these challenges.

Static Scheduling – Instead of event-driven simulation, Mamba statically schedules update blocks. While static scheduling has been shown to improve the performance of C++-based simulation frameworks [PMT04, GTBS13], we argue that static scheduling is particularly important in HGSFs that run on tracing JITs for two reasons: (1) static scheduling avoids bridges due to data-dependent checks on every signal assignment; and (2) static scheduling paves the way for using additional techniques to increase the length of each trace. The Mamba execution semantics require each update block to be executed exactly once in each cycle. This enables a static fixed-order linear schedule to be generated at elaboration time. We leverage the sensitivity information to schedule the update blocks correctly: an update block that writes x should be scheduled before all blocks that read *x*. We use a topological sort to serialize the dependency graph into a total order of blocks. The topological sorting can succeed only if the directed graph is acyclic (DAG). Thus designers must not create inter-dependencies between combinational blocks. Note that an inter-dependency does not always imply a combinational loop; two different variables can be read and written in two blocks. In the example component in Figure 5.1, there are no combinational dependencies, so any ordering of the update blocks is valid.

In Figure 5.2, the `tick_static` function (lines 15–18) shows an example static schedule of the example component. The inner loop simply iterates over the static schedule. Note that this does not change the meta-trace patterns in Figure 5.3(a); this simply changes the way in which the corresponding update blocks are updated. Table 5.1 shows that this approach improves the performance by 1.1–13× over event-driven simulation. One drawback of static scheduling is that all update blocks are executed regardless of activity. In event-driven scheduling, if the combinational update block does not have the signals in its sensitivity list change, it will not be called. This enables higher simulation speeds using event-driven scheduling for hardware that is often idle. For example, the performance difference between a greatest common divisor unit which is active 10% vs. 100% of the time in event-driven PyMTL is almost 7×, while the same performance gap is only 1.5× in Mamba due to static scheduling.

Schedule Unrolling – Static scheduling makes it possible to eliminate the pathological chain-of-bridges pattern in the inner loop by unrolling this loop into a sequence of update block calls. The `tick_static_unroll` function in Figure 5.2 (lines 20–25) shows how unrolling might be implemented. To unroll the schedule, the Mamba framework generates Python code with unrolled update block calls, and executes this generated code. Being able to programmatically generate code on the fly at the application level and execute this with high efficiency using JIT is a testament to the flexibility of dynamic languages. Table 5.1 shows that this improves performance by 1.2–9× compared to static scheduling without inner-loop unrolling for the divider and the 1-core design. Figure 5.3(b) illustrates how static scheduling and schedule unrolling get rid of the chain-of-bridges pattern but increase the overall trace length. As each update block body is unrolled and inlined, the branches in these update blocks cause a separate trace to be compiled. Furthermore, in each of the original trace and bridges, subsequent branches may also cause additional bridges to be compiled. The branch in line 11 in Figure 5.1 creates two traces, A and A'. As `update_b` is also unrolled, its two branches in lines 16 and 17 cause two additional branches for *each* existing trace. Because of this, `update_c` and `update_d`, despite neither of them containing any branches, are compiled 6 times. These redundant compilations hurt the memory usage, instruction cache hit rates, and increase the compilation and optimization overhead.

Heuristic Topological Sort – Unfortunately, schedule unrolling can create an *exponential number of bridges* due to data-dependent control flow within each update block. Every code path permutation due to control flow in update blocks (A/A' and B/B'/B" in Figure 5.3(b)) can create a new bridge. In other words, schedule unrolling introduces a new pathological pattern that can lead to serious performance degradation in larger designs (see 32-core in Table 5.1). To address this problem, we observe that there are multiple valid topological sorts for any given DAG, and each ordering can produce different guard/bridge behavior in our scenario. For example, `tick_static_unroll_toposort` in Figure 5.2 (lines 27–32) and Figure 5.3(c) illustrate an ordering with fewer guards and bridges (and a smaller instruction-cache footprint) compared to Figure 5.3(b). We use a heuristic to schedule update blocks with potentially more guards as late as possible. The stack used in the topological sort is replaced with a priority queue where each update block's priority is the number of `if/elif` statements in that block counted using AST self-parsing. This approach can reduce the number of redundantly compiled update blocks. `update_c` and `update_d` are now only compiled once, but `update_b` is still compiled twice. Table 5.1 shows that this can improve the performance by 10–30% over basic schedule unrolling.

Trace Breaking – A large number of guards and bridges is still possible in more complex designs. To further control the number of guards and bridges, we use Python-level JIT hints to break long traces into multiple smaller traces. These application-level hints are provided by PyPy to control the JIT compilation process, and they can be used to prevent tracing in certain parts of the application. During the topological sort, we pack update blocks into a *meta-update block*. A meta-update block is a sequence of one or more update blocks that do not include any `if/elif` statements followed by a final update block which does include an `if/elif` statement. A meta-update block ends with a trace-breaking hint. This technique essentially limits the number of `if/elif` statements within any given trace. The function `tick_static_unroll_toposort_break` in Figure 5.2 (lines 34–45) shows how meta-update blocks are formed and the application-level JIT hint to force a trace break. As Figure 5.3(d) shows, this approach can entirely eliminate redundant compilations. Table 5.1 shows this technique has a more significant impact on larger designs, e.g., improving performance by 5× for the 32-core design over heuristic topological sort.

Block Consolidation – Despite the techniques described above, the size of JIT-compiled code scales with the design size due to the nature of JIT compilation: the same update block from different instances is JIT-compiled individually. This problem is less prominent in static languages because different instances of the same module will likely reuse the same compiled assembly code. We observe that modular replication and composition is very common in RTL design (e.g., a chip with 16 homogeneous cores is more common than 16 disparate cores). Block consolidation is a new technique that deduplicates different instances of an update block in a JIT trace. We modify the topological sort to identify different instances of the same update block and to then schedule these instances together. We use a hash of the update block source code to identify when they are the same. We group these update blocks into a new nested loop that iterates over these different instances by calling the same update block with different parameters in each iteration. Table 5.1 shows that large designs can significantly benefit from block consolidation, e.g., improving performance by 28$\times$ for the 32-core design over trace breaking.

5.2.2 HGSF-Aware JIT

The previous section described techniques to improve the performance of an HGSF when using a general-purpose meta-tracing JIT. In this section, we describe two techniques to improve performance by making the JIT specialized for the HGSF.

Meta-Tracing the Performance-Critical Constructs – Although the PyPy JIT compiler can run arbitrary Python code, native Python constructs may not be the best fit for HGSFs. For example, fixed-bit-width data types are used extensively in HGSFs, but they are not natively supported by Python. HGSF designers must emulate slicing and two's complement arithmetic using integer arithmetic. This increases warm-up time, requires redundant arithmetic operations, and creates excessive bridges due to dynamic type casting. We implement a fixed-bit-width data type in RPython as a proof of concept. Other performance-critical constructs (e.g., byte-addressable memory) can also be implemented in RPython. The key is the meta-tracing approach that enables writing Python-like code exactly once. We exploit the invariant that the bit-width of a signal does not change during simulation; RPython enables annotating the bitwidth as immutable. We are also able to directly manipulate the underlying integer arrays at the RPython level. These specializations significantly eliminate potential bridges. Table 5.1 shows that this technique improves the performance by an additional 1.5–3.5$\times$ on top of the JIT-aware HGSF techniques.

Support for Huge Loops – The techniques described in Section 5.2.1 improve performance but also often increase the total size of all traces. PyPy's VMProf tool is only useful for identifying Python-level bottlenecks, so we use the Linux *perf* tool to identify the microarchitectural implications of these larger instruction cache footprints. Experiments show that for the 8-core (1-core) simulation in Section 5.3, 3% (0.2%) of all instruction fetches incur an instruction TLB load, among which 22% (2.6%) are iTLB misses. The need for larger TLB reach motivates us to modify PyPy to allocate 2 MB huge pages for traces and to fall back to 4 KB pages if Linux's huge-page support is unavailable. As a proof of concept, the removal of excessive iTLB accesses (confirmed by *perf*) improved the performance of the 32-core design by 10% as shown in Table 5.1.

5.3 Case Study: RISC-V Multicore

We present an apples-to-apples simulation performance comparison of 1–32 RV32IM [AP14] five-stage cores implemented in-house in Verilog, PyMTL, and Mamba using a structural datapath and pipelined control unit. The cores run a parallel matrix multiplication application kernel using a lightweight parallel runtime. We simulate Verilog with a fast commercial verilog simulator, CVS[1], Icarus [ica], and Verilator [ver14], and we use PyPy for PyMTL, PyMTL-CSim, and Mamba. The multi-core does not include caches nor an interconnection network and is simulated with a behavioral test memory implemented in Verilog for CVS and Icarus, C++ for Verilator, and Python for PyMTL and Mamba. ASIC synthesis results show that each core can be implemented in around 10 K gates. This design is sufficient for exploring the scalability of various hardware development frameworks, and more complex system-on-chip designs are left as future work. The simulation platform includes an Intel E3-1240 v5 processor and 32 GB DDR4-2400 memory running Ubuntu 14.04 Server, gcc-4.8.5, PyPy-5.8, Verilator-3.876, and Icarus-11.0.

Compilation/Warmup – Figure 5.4(a) and (b) reflect the iterative development cycle for simulating a specific number of instructions. This includes all overheads: CVS, Icarus, Verilator, and PyMTL-CSim compile times; PyMTL and Mamba elaboration times; and PyMTL and Mamba JIT warmup times. Intuitively, the leftmost points (i.e., short simulations) are affected the most by these overheads. Overall, CVS and Icarus have relatively low compilation overhead (1–2 s for 1-core, 3 s for 32-core), whereas Verilator has larger compilation overhead (4–5 s for 1-core, 130 s for 32-core).

[1]Tool vendor anonymized due to license agreement.

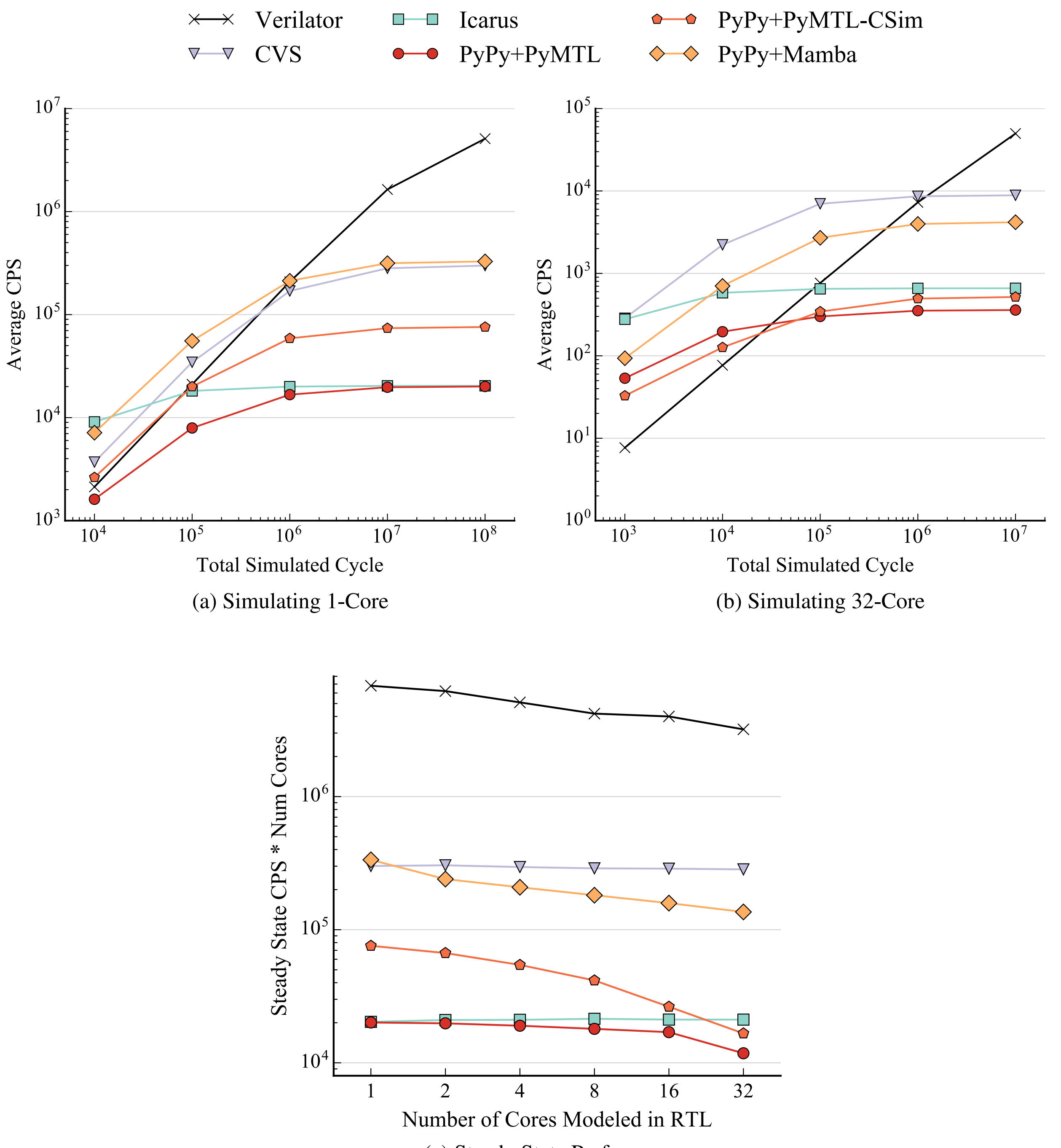


Figure 5.4: Performance of Simulating RISC-V Multicore – Each point in (a) and (b) is the average simulated cycle per second (CPS) taking compilation overhead into account; (c) captures the scalable performance: steady state CPS multiplied by number of simulated cores.

PyMTL and Mamba have short elaboration times for one core (<1s) but longer elaboration times for 32 cores (6-8s). The JIT warmup overhead is difficult to quantify; both PyMTL and Mamba warm up within at most 10^5 simulated cycles, and the absolute warm-up time is shorter in Mamba compared to PyMTL.

Performance – When simulating a 1-core system, Mamba executes 332K CPS which is slightly faster than CVS and significantly faster than the other frameworks. Mamba's 1-core performance is equivalent to 148K committed instructions per second. When simulating a 32-core system Mamba is 2.1× slower than CVS but again significantly faster than the other frameworks. Overall these results demonstrate that Mamba nearly matches the performance of CVS for both small and large designs for both short and long simulations. While Verilator can achieve impressive performance for long simulations, it can be difficult to amortize Verilator's long compile times for short simulations potentially precluding using Verilator in agile test-driven development.

Scalability – Figure 5.4(c) summarizes the steady-state performance of all frameworks with a gradually increasing number of simulated cores. We multiply the simulated cycles per second by the number of cores to reflect the simulation performance scaling with the size of design. A flat line indicates perfect scalability (i.e., a 2× larger design results in a 2× reduction in CPS). CVS and Icarus have good scalability, whereas Verilator appears to be less scalable. The source code size generated by Verilator scales up linearly with the number of cores, potentially harming the quality of C++ compilation. Mamba is faster than CVS at 1-core, and only 2× slower at 32-core. PyMTL scales better than PyMTL-CSim and Mamba, but its absolute performance is relatively low.

5.4 Conclusion

We have presented Mamba, JIT-aware HGSF and HGSF-aware JIT techniques that significantly improve the performance of modern RTL design methodologies that take advantage of embedded DSLs in dynamic languages. The Mamba techniques will be incorporated into the next generation of PyMTL. Mamba is able to reach the performance of CVS, and come within an order of magnitude of Verilator for long-running simulations. Even more impressive is Mamba's performance for shorter running simulations as the JIT is able to warm up much quicker than Verilator can translate models to C++. Mamba shows that meta-tracing JITs can help accelerate RTL simulations while also allowing high-productivity design experience as highlighed in PyMTL.

PART II
BUILDING HARDWARE ACCELERATORS FOR META-TRACING JIT VMS

The second part of the thesis focuses on developing hardware specialization to accelerate meta-tracing JIT VMs. Meta-tracing JIT virtual machines for dynamic languages contain many different components including an interpreter, library components, garbage collector, a meta-interpreter to generate the trace, compiler optimization passes, and a backend that generates the JIT-compiled code. Furthermore, the RPython framework contains many different abstraction layers and languages, from interpreter code that is written in Python to the translation toolchain that converts the VM to C, and the JIT-compiled code that is generated on the fly. As a result, it is non-obvious which components are critical to the performance of dynamic languages executing on a meta-tracing JIT VM. Chapter 6 performs a comprehensive cross-layer workload characterization of meta-tracing JIT VMs, looking at application-, framework-, interpreter-, JIT-, and microarchitecture-level quantitative analysis of the VM components. Chapter 7 explores further hardware acceleration ideas specifically in the JIT-compiled code. I identify a promising value pattern with respect to object dereferences. Object dereferences are a common occurrence due to both application patterns typically used in dynamic languages and the way meta-tracing automatically generates JIT-compiled code using the interpreter. I quantify object dereference locality in meta-tracing JIT VMs and propose a promising hardware specialization approach called skippable sub-traces to reduce dynamic instruction count.

CHAPTER 6
META-TRACING WORKLOAD CHARACTERIZATION

Meta-tracing JIT VMs contain many different components and it is non-obvious which components are critical to the performance. Before proposing a new hardware acceleration solution, it is important to know which components can have bottlenecks. This chapter presents a thorough cross-layer workload characterization of meta-tracing JIT VMs and provides a first attempt in answering a number of important performance-related questions using data.

6.1 Introduction

Dynamic programming languages are growing in popularity across the computing spectrum from smartphones (e.g., JavaScript for mobile web clients), to servers (e.g., Node.js, Ruby on Rails), to supercomputers (e.g., Julia for numerical computing). Table 2.1 shows many of the top-ten most popular programming languages are now dynamic. Dynamic languages typically include: lightweight syntax; dynamic typing of variables; managed memory and garbage collection; rich standard libraries; interactive execution environments; and advanced introspection and reflection capabilities. The careful use of these features can potentially enable more productive programming.

However, the very features that make dynamic languages popular and productive also result in lower performance. These languages traditionally use interpreters to implement a virtual machine that closely aligns with the language semantics, but interpreted code can be orders-of-magnitude slower than statically compiled code. To address this performance/productivity gap, dynamic languages are using just-in-time (JIT) optimizing virtual machines (VMs) to apply traditional ahead-of-time compiler techniques at run-time [BCFR09, BT15, WWW$^+$13, WWS$^+$12, Höl13, GES$^+$09, spi, jul, v8]. Such JIT optimizations include removing the (bytecode) fetch and decode overhead, generating type-specialized code for the observed types, (partial) escape analysis [SWM14, BCF$^+$11a], constant propagation, and dead-code elimination. It is well known that developing state-of-the-art JIT-optimizing VMs is very challenging due to the overall complexity (e.g., the need for profiling, recording a trace, compiling, deoptimizations on type misspeculation, garbage collection, etc.), performance requirements (e.g., the JIT optimization process itself must be fast since it is on the critical path), and development process (e.g., debugging dynamic code generation). With many programmer-decades of engineering effort, some JIT-optimizing VMs (e.g., Google V8 [Höl13, v8], Mozilla SpiderMonkey [spi]) can begin to achieve performance that is within 10$\times$ of statically compiled code.

Unfortunately, many emerging and experimental languages simply cannot afford the effort required to implement a custom state-of-the-art JIT-optimizing VM. This has motivated work on meta-JIT optimizing VMs (or “meta-JITs”) which abstract the language definition from the VM internals such that language implementers need not worry about the complexity typically associated with JIT optimizations. There are currently two production-ready meta-JITs: the Truffle framework for rapidly developing method-based JIT-optimizing VMs [WWH+17, WWW+13, WWS+12] and the RPython framework for rapidly developing trace-based JIT-optimizing VMs [BCFR09, BT15] (see [MD15] for a detailed comparison of these frameworks). The *Truffle framework* enables language implementers to define abstract-syntax-tree- (AST-) based interpreters for their language and also specify JIT-optimization opportunities to the Graal compiler. Truffle automatically identifies “hot” target-language methods and then applies the previously specified JIT optimizations along with aggressive partial evaluation before targeting the HotSpot JVM. The *RPython framework* enables language implementers to build AST- or bytecode-based interpreters in a high-level language. An interpreter communicates to the framework its dispatch loop, target-language loops, and additional run-time hints. RPython automatically identifies “hot” target-language loops and then generates a trace, optimizes the trace, and lowers the trace into assembly. Meta-JITs can significantly reduce the effort involved in building JIT-optimizing VMs, and so it is not surprising that many language interpreters are now either using or experimenting with meta-JITs (e.g., Python [BCFR09], Ruby [top15, Sea15, trub], JavaScript [WWS+12], R [trua], Racket [BPSTH14], PHP [hip15], Prolog [BLS10], Smalltalk [BKL+08]). To narrow the scope of this work, we focus on the RPython meta-tracing JIT, and Section 6.2 provides additional background on this framework.

We anticipate the trend towards meta-JITs will continue for new, research, and domain-specific languages, and this motivates our interest in performing a multi-language workload characterization of the RPython meta-tracing JIT. Section 6.3 describes our baseline characterization methodology. One of the key challenges in performing such a characterization is the many layers of abstraction used in a meta-tracing JIT including: the target dynamic programming language; the target language AST or bytecode; the high-level language used to implement the interpreter; the intermediate representation (IR) used in the meta-trace; and the final assembly instructions. In Section 6.4, we describe a new cross-layer characterization methodology that enables inserting cross-layer annotations at a higher layer, and then intercepting these annotations at a lower layer. In Section 6.5, we use this new cross-layer methodology to characterize a diverse selection of benchmarks written in Python and Racket at the application, framework, interpreter, JIT-IR, and microarchitecture level. In Section 6.6, we use this characterization data to answer the following nine key questions:

1. Can meta-tracing JITs significantly improve the performance of multiple dynamic languages?
2. Does generating and optimizing traces in a meta-tracing JIT add significant overhead?
3. Does deoptimization in a meta-tracing JIT consume a significant fraction of the total execution time?
4. Does garbage collection in a meta-tracing JIT consume a significant fraction of the total execution time?
5. Is all JIT-compiled code equally used in a meta-tracing JIT?
6. Does a meta-tracing JIT need to spend most of its time in the JIT-compiled code to achieve good performance?
7. What fraction of the time in the JIT-compiled code is overhead due to the meta-tracing JIT?
8. What is the microarchitectural behavior (e.g., instruction throughput, branch prediction) of JIT-compiled code?
9. Why are meta-tracing JITs for dynamic programming languages still slower than statically compiled languages?

6.2 Background on Meta-Tracing and RPython

This section supplements the background provided in Chapter 2 from the perspective of workload characterization. The RPython toolchain uses a novel approach where the tracing JIT compiler is actually a *meta-tracing JIT compiler*, meaning that the JIT does not directly trace the application, but instead traces the *interpreter* interpreting the application. The interpreter is written in RPython, a statically typed subset of Python. The interpreter uses a dispatch loop to continuously fetch a quantum of execution (e.g., bytecode) and dispatch to a corresponding execution function. The RPython framework has its own standard library (similar to Python's standard library) and an API so that the language implementer can inform the framework of the interpreter's program counter,

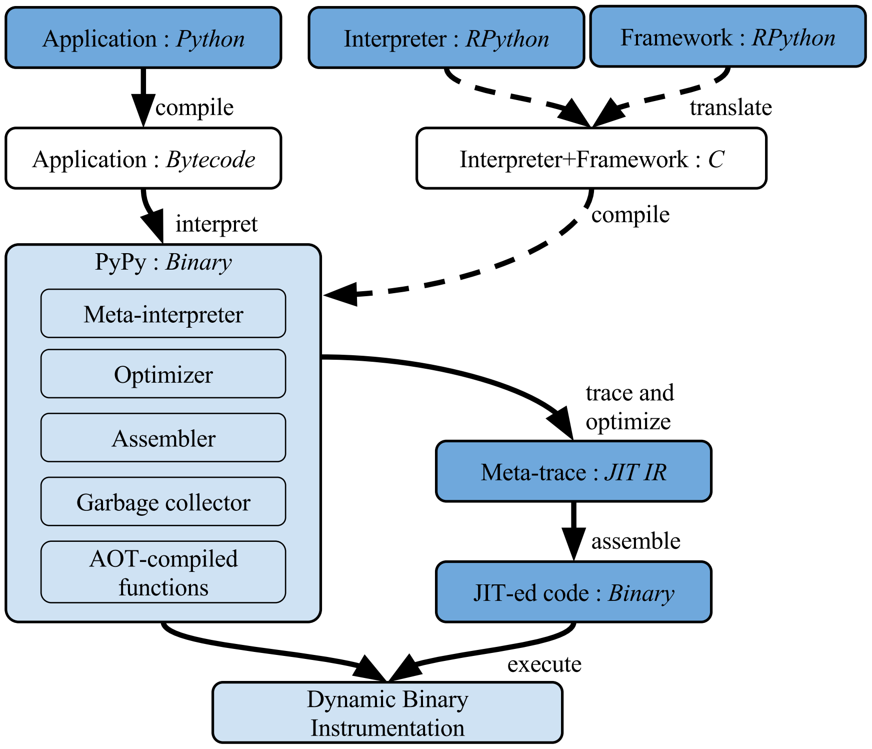


Figure 6.1: RPython Framework – Dashed lines indicate ahead-of-time operations; solid lines indicate execution-time operations. The language each block is written in is indicated after the colon. Dark blue blocks are where we can insert cross-layer annotations, and light blue blocks are where we can intercept cross-layer annotations.

dispatch loop, and application-level loops. The framework also supports hints to indicate which variables can be treated as constants in the trace, which interpreter-level functions are pure, and when type-specialized versions of functions should be generated.

Since RPython is a proper subset of Python, interpreters written in RPython can be executed using a standard Python interpreter. However, for good performance, these interpreters are automatically translated into C using the RPython translation toolchain (see dashed lines in Figure 6.1). At run-time, the application (e.g., Python code) is compiled into bytecode, and the bytecode executes on the C-based interpreter (see solid lines in Figure 6.1). When the interpreter reaches an application-level loop, the framework increments an internal per-loop counter. Once this counter exceeds a threshold, the execution is transfered to a meta-interpreter. The meta-interpreter builds the meta-trace by recording the operations performed by the interpreter until the application-level loop is completed. The trace is then passed on to a JIT optimizer and assembler before initiating native execution. A meta-trace includes guards that ensure the dynamic conditions under which the meta-trace was optimized still hold (e.g., the types of application-level variables remain constant).

If a guard fails or if the optimized loop is finished, the JIT returns control back to the C-based interpreter using a process called *deoptimization*. Deoptimization transforms the intermediate state in a JIT-compiled trace to the precise state required to start execution of the interpreter. If a guard fails often, it is converted into a *bridge*, which is a direct branch from one JIT-compiled trace to another, separately JIT-compiled trace. More on the RPython toolchain can be found in Chapter 2 and [SB12, Bol12, AACM07].

Meta-tracing JITs have some important differences compared to traditional tracing JITs. Tracing JITs trace the execution of the target program directly (e.g., recording the trace of executed bytecodes), while meta-tracing JITs trace the execution of the interpreter as it executes the target program. These extra levels of abstraction can potentially result in increased overhead during tracing, deoptimization, garbage collection, and JIT-compiled execution. Also note that the meta-interpreter opportunistically inlines interpreter-level function calls. However, if these functions contain loops with data-dependent bounds, they are excluded from the trace to avoid numerous guard failures. These functions are ahead-of-time- (AOT-) compiled (and so not dynamically optimized) and then called from within the JIT-compiled code. While traditional tracing JITs may also include calls to the JIT framework from within JIT-compiled code, there is the potential for many more calls in a meta-tracing JIT because such calls are particularly easy to use in RPython.

6.3 Baseline Characterization Methodology

To characterize meta-tracing JITs across multiple languages, we will study Python (a well-known, general-purpose programming language) and Racket (a general-purpose programming language in the Lisp-Scheme family). As a baseline, we will use the "reference" interpreters for both Python (i.e., CPython) and Racket. Technically, Racket does not use an interpreter but instead uses a custom JIT-optimizing VM. Both Python and Racket also have highly optimized meta-tracing JITs implemented using the RPython framework. For Python, we will use the very popular PyPy meta-tracing JIT [BCFR09] and for Racket, we will use the Pycket meta-tracing JIT [BPSTH14]. For each language, we will also explore an interpreter generated from the RPython translation toolchain but without the meta-tracing JIT enabled as another example of a traditional interpreter. Finally, we also include results for C/C++ implementations of some benchmarks to provide a reference for statically compiled languages.

We use benchmarks from two suites: the PyPy Benchmark Suite [pypb] and the Computer Languages Benchmarks Game (CLBG) [Gou]. We use the PyPy Benchmark Suite because it is widely used for benchmarking Python programs [BCF$^+$11a, SB12]. We use the CLBG to compare the performance of different languages and interpreters on highly optimized implementations of the same benchmarks. The benchmarks in the PyPy Benchmark Suite are single-threaded and single-process, whereas many implementations in the CLBG make use of multi-programming and multi-threading. CPython and the RPython framework currently make use of a Global Interpreter Lock (GIL) [Bea10], preventing parallelism in multi-threaded programs. Because the focus of the chapter is on the performance characterization of meta-tracing JITs, and not of GILs and parallelism, we restrict all benchmarks, including many parallel implementations in the CLBG, to use a single hardware thread. Because CLBG provides multiple implementations of the same benchmark and language, we pick the fastest implementation for each benchmark and language combination. Due to some missing features of the Racket language in Pycket, a number of the CLBG benchmarks did not execute correctly.

When characterizing at the JIT IR level, we make use of the PyPy Log facility, which is part of the RPython framework. The PyPy Log contains information about each JIT-compiled trace including the bytecode operations, JIT IR nodes, and assembly instructions contained in each trace, along with the number of times each trace was executed. Enabling the PyPy Log slightly degrades performance (<10%), so we disabled this feature when comparing the overall execution time of the interpreters.

We use two mechanisms to collect microarchitectural measurements through performance counters. The first uses the performance application programming interface (PAPI) to record performance counters on certain cross-layer annotations [MBDH99]. We implemented this mechanism in the RPython-based interpreters. To compare microarchitectural characteristics of meta-tracing to other interpreters and statically compiled code, we use Linux's `perf` tool to periodically read the performance counters.

6.4 Cross-Layer Characterization Methodology

The baseline characterization methodology described in Section 6.3 can enable initial analysis, but the many layers involved in a meta-tracing JIT make it difficult to gain insight into cross-layer interactions between: the target dynamic programming language; the target language AST or bytecode; the interpreter including the RPython standard library; the JIT IR used in the meta-trace; and the final assembly instructions. In this section, we describe a new cross-layer characterization methodology based on using *cross-layer annotations*.

Cross-layer annotations are a unified mechanism to annotate events of interest at one level of meta-tracing execution, and collect these annotations at different level. For instance, a Python application might annotate when a particular function is called, the Python interpreter might annotate every time the dispatch loop is executed, the RPython framework might annotate when loops are being traced or when garbage collection occurs, and generated machine code from the JIT compiler might annotate when a particular assembly sequence corresponding to a particular IR node is being executed. Figure 6.1 shows the blocks where the cross-layer annotations can be inserted in dark blue.

These annotations can be collected at different levels. At the assembly instruction level, annotations can be observed by using a carefully selected instruction which does not change the program behavior but also includes a tag to indicate the corresponding annotation. Our methodology uses the `nop` instruction in the x86 ISA. Although the `nop` instruction supports the usual addressing modes of the x86 ISA, the architecture only considers the opcode and ignores the corresponding address. Our methodology uses a unique address to serve as the tag for each cross-layer annotation. Other ISAs can use other instructions or sequences of instructions to achieve a similar effect (e.g., `add r1, r1, #145`; `sub r1, r1, #145` in ARM). The execution target that executes these machine instructions (e.g., a dynamic instrumentation tool, an ISA simulator, or a soft-core processor on an FPGA) can perform some action when one of these cross-layer annotations is executed. Cross-layer annotations can also be collected at higher levels. For example, the timestamps or microarchitectural measurements can be logged to a file every time a cross-layer annotation is called. Figure 6.1 shows the blocks where the cross-layer annotations can be collected in light blue.

In this chapter, we have modified the RPython framework and inserted cross-layer annotations at various events of interest in the framework (e.g., when a minor garbage collection starts and ends, when tracing starts and ends, when execution starts on the JIT-compiled code, etc.). This allows us to know exactly what the framework is doing at a given point of time. We can use this information to get detailed breakdowns of time spent in different parts of the framework. We also added cross-layer annotations at the interpreter level at the beginning of the dispatch loop. This allows us to have an independent measure of "work" (e.g., number of bytecodes in PyPy) regardless of whether the interpreter is being used (if the JIT is off or has not warmed up yet), the tracing interpreter is being used, or the JIT-compiled code is being executed. This enables precisely finding the JIT warmup break-even point. Finding the break-even point using other techniques is likely very difficult because counting the number of bytecodes executed in the JIT would likely introduce significant performance overheads that would skew the results. We added application-level APIs to our interpreters so that cross-layer hints can also be emitted from the application level. Finally, we can also emit cross-layer annotations when each JIT IR node is lowered to assembly instructions. This enables tracking the connection between traces, JIT IR nodes, and assembly instructions. Each cross-layer annotation can be enabled/disabled from the command line.

We use the Pin dynamic binary instrumentation tool [LCM+05] as the primary mechanism for intercepting cross-layer annotations. We have developed a custom PinTool that detects the `nop` instructions and can track information on the phase, the bytecode execution rate, AOT-compiled functions, and JIT IR nodes.

6.5 Cross-Layer Workload Characterization

This section presents cross-layer workload characterization studies that will help us answer the key questions regarding meta-tracing JITs.

6.5.1 Application-Level Characterization

We first compare the overall application performance of dynamic languages running on different VMs. Table 6.1 compares the application performance of the PyPy Benchmark Suite using CPython, PyPy without a meta-tracing JIT, and PyPy with a meta-tracing JIT. We can see that CPython is consistently faster than PyPy with its meta-tracing JIT disabled for almost all of the benchmarks, usually by 2× or more. The first reason for this is because CPython is written directly in C, whereas PyPy is written in a high-level language (RPython) translated to C. The other reason is because CPython is designed to only be an interpreter, and thus it includes some modest interpreter-focused optimizations. The performance benefit of PyPy with the meta-tracing JIT over CPython is much more varied and usually much higher: from 0.7–51×.

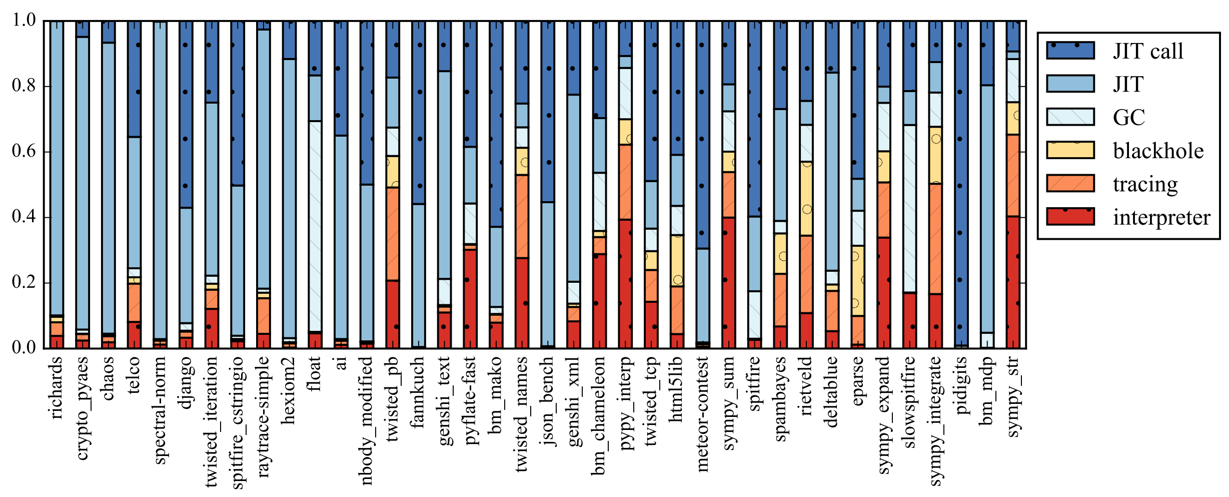


Figure 6.2: RPython Framework Breakdown – Shows the breakdown of time spend in various components of the RPython framework.

To compare different languages, Table 6.2 shows the overall execution time of the CLBG benchmarks. Similar trends hold here between PyPy with the meta-tracing JIT enabled and CPython, except for a few cases (`chameneosredux`, `pidigits`, `revcomp`) where CPython performs much better. These Python programs use external libraries, which are often written in C using an API that exposes CPython implementation details. There is ongoing work on a PyPy C-compatibility layer which could enable similar performance benefits. We see that the other RPython-based meta-tracing JIT, Pycket, has similar performance as Racket with a range of 0.3–2×. This is due to: (1) Pycket is less mature compared to PyPy; and (2) unlike CPython, Racket uses a custom JIT-optimizing VM. Racket- and Python-language implementations, even with a meta-tracing JIT, tend to perform very poorly compared to C and C++.

6.5.2 Framework-Level Characterization: Phases

Tracing JITs typically have different phases inherent to the way they execute and optimize code. Initially, execution starts in the *interpreter phase*. When hot loops are detected, these loops are traced and compiled in the *tracing phase*. The JIT-compiled code is executed during the *JIT phase*. Occasionally, there are calls to AOT-compiled functions in the runtime from JIT-compiled code, which is the *JIT call phase*. Finally, there is the *GC* phase for garbage collection and the *blackhole* phase deoptimization (due to the “blackhole interpreter” used to implement deoptimization in RPython).

	CPython			PyPy w/o JIT				PyPy with JIT			
Benchmark	**t (s)**	**IPC**	**M**	**t (s)**	**vC**	**IPC**	**M**	**t (s)**	**vC**	**IPC**	**M**
richards	0.2	1.65	5.9	0.5	0.5	1.32	6.4	0.004	51.2	1.38	3.5
crypto_pyaes	2	1.94	3.1	4	0.4	1.55	3.2	0.06	30.2	1.62	0.8
chaos	0.3	1.49	5.5	0.7	0.4	1.03	6.7	0.01	27.2	1.31	1.9
telco	0.9	1.24	7.4	2	0.4	0.88	7.3	0.03	27.1	1.11	4.0
spectral-norm	0.3	1.93	3.3	0.9	0.3	1.44	3.6	0.01	25.9	1.90	0.8
django	0.7	1.24	5.7	1	0.5	0.88	6.6	0.04	18.2	1.37	2.5
twisted_iteration	0.09	1.41	4.5	0.4	0.2	0.95	7.1	0.006	15.0	1.27	0.8
spitfire_cstringio	10	1.92	1.4	9	1.1	1.46	3.0	0.9	11.4	2.00	0.5
raytrace-simple	2	1.54	5.4	4	0.5	1.13	5.8	0.2	10.4	1.22	2.7
hexiom2	149	1.88	2.5	442	0.3	1.38	4.2	14	10.1	1.91	1.2
float	0.4	1.62	2.5	0.8	0.5	1.22	4.6	0.05	7.1	1.38	2.9
ai	0.3	1.44	3.7	1	0.3	1.05	4.7	0.04	7.0	1.79	1.6
nbody_modified	0.3	2.06	2.9	0.9	0.3	1.58	3.0	0.04	6.9	1.50	1.1
twisted_pb	0.05	1.12	7.9	0.1	0.4	0.86	7.0	0.007	6.4	0.68	4.1
fannkuch	1	1.70	3.8	2	0.6	1.36	6.5	0.2	5.2	1.59	5.5
genshi_text	0.1	1.29	6.2	0.4	0.3	0.94	7.1	0.02	5.2	1.30	1.8
pyflate-fast	2	1.68	4.4	4	0.5	1.29	5.6	0.4	4.8	1.62	2.4
bm_mako	0.1	1.46	2.3	0.3	0.4	0.89	5.1	0.02	4.8	1.41	2.3
twisted_names	0.008	0.74	13.9	0.02	0.5	0.67	9.5	0.002	4.1	0.51	9.4
json_bench	3	1.54	4.6	31	0.1	1.17	6.2	0.9	3.9	1.91	0.7
genshi_xml	0.2	1.11	7.6	0.8	0.3	0.78	8.6	0.06	3.9	1.09	1.4
bm_chameleon	0.07	1.35	5.4	0.2	0.4	1.05	5.6	0.02	3.5	1.39	2.5
pypy_interp	0.3	1.15	7.0	0.6	0.5	0.89	6.5	0.1	3.3	0.91	6.4
twisted_tcp	0.6	0.68	10.3	1	0.5	0.54	9.2	0.2	3.0	0.48	3.4
html5lib	11	0.93	9.7	27	0.4	0.77	7.0	4	2.5	0.89	4.6
meteor-contest	0.2	1.51	7.0	0.6	0.4	1.32	3.4	0.1	2.4	1.64	3.9
sympy_sum	1	1.25	5.6	5	0.3	0.80	6.6	0.6	2.3	0.99	5.6
spitfire	5	1.82	2.4	12	0.4	1.37	2.7	2	2.1	1.55	1.1
spambayes	0.2	1.59	4.3	0.6	0.4	1.35	4.2	0.1	2.0	0.99	6.4
rietveld	0.6	0.99	9.1	1	0.5	0.76	7.8	0.3	1.8	0.76	8.4
deltablue	0.02	1.74	3.5	0.05	0.4	1.25	4.2	0.01	1.7	0.98	6.1
eparse	0.8	1.35	5.5	1	0.6	0.96	6.3	0.5	1.5	0.77	5.8
sympy_expand	1	1.17	7.1	3	0.4	0.83	6.7	0.8	1.4	0.86	6.4
slowspitfire	0.4	1.86	2.1	1	0.4	1.52	2.2	0.3	1.3	1.39	0.5
sympy_integrate	4	1.30	5.5	15	0.2	0.88	6.3	3	1.2	0.86	6.8
pidigits	12	2.45	0.1	11	1.1	1.83	0.1	10	1.1	1.84	0.1
bm_mdp	10	1.34	8.6	70	0.2	1.19	5.2	10	1.1	1.39	2.1
sympy_str	0.5	1.14	7.7	1	0.4	0.83	6.9	0.7	0.7	0.88	7.2
Average		1.46	5.4		**0.5**	1.10	5.6		**1.7**	1.27	3.4

Table 6.1: PyPy Benchmark Suite Performance – Overall execution times ordered by PyPy with JIT speedup over CPython. vC = speedup compared to CPython. IPC = instructions per cycle. M = branch misses per 1000 instructions.

	C	CPython		PyPy		Racket		Pycket	
Benchmark	**IPC**	**Sdn**	**IPC**	**Sdn**	**IPC**	**Sdn**	**IPC**	**Sdn**	**IPC**
binarytrees	2.16	37	1.95	4.5	1.37	5.7	1.80	11	1.26
chameneosredux	1.19	87	1.13	1374	0.86	111	0.99	–	–
fannkuchredux	1.16	89	1.85	25	1.43	13	1.97	7.2	1.16
fasta	1.89	8.3	1.93	5.6	1.23	2.0	1.09	2.1	1.09
knucleotide	1.74	15	1.69	8.8	1.65	4.4	2.20	–	–
mandelbrot	1.99	115	2.23	29	1.29	9.0	1.52	6.7	1.21
meteor	1.18	78	1.78	30	1.04	12	1.63	31	1.06
nbody	2.23	97	2.16	12	1.33	5.3	1.68	2.8	1.20
pidigits	1.65	1.8	1.48	5.1	0.96	13	0.93	7.4	1.89
regexdna	1.36	3.8	1.78	3.1	1.12	6.7	2.01	–	–
revcomp	1.43	4.0	1.96	6.8	1.35	5.1	2.10	4.0	1.21
spectralnorm	1.20	104	1.97	10	1.25	6.4	1.56	4.2	1.16
threadring	1.03	20	0.97	16	0.94	16	0.82	–	–

Table 6.2: CLBG Performance – IPC and slowdowns (Sdn) compared to C/C++. Meta-tracing JIT is enabled for PyPy and Pycket.

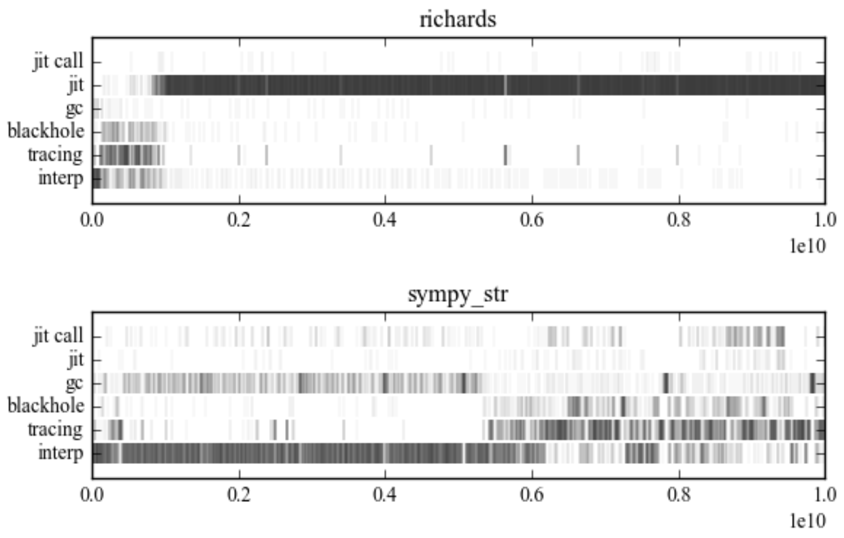


Figure 6.3: Framework Phases in Time – Each line indicates which phase the framework currently is in during the first 10 billion instructions of the best-performing (`richards`) and worst-performing (`sympy_str`) benchmarks.

Using cross-layer annotations and a custom PinTool that intercepts them, we can tease apart how much of the execution time is spent in each of these phases. Figure 6.3 shows the phases for the best- and worst-performing (compared to CPython) benchmarks. As expected, the framework initially spends most of its time in the interpreter, tracing, and blackhole phases until the meta-tracing JIT warms up, then time in JIT phase dominates. Interestingly, garbage collection is used more heavily before the JIT phase. This is most likely due to escape analysis in the JIT which removes many object allocations. Figure 6.2 shows the breakdown of time spent in each phase by benchmark. For some benchmarks, the *JIT* and *JIT call* phases dominate the execution (e.g., `ai`, `json_bench`), while others spend most of their time in the interpreter (e.g., `sympy_str`). With the exception of *blackhole*, every different phase dominates the execution time of at least one benchmark. This shows that none of these phases can be ignored from an optimization perspective, and aggressive improvements in one of these phases unfortunately translates to modest improvements of execution times on average. Figure 6.5 compares the phase breakdown for the meta-tracing JITs on the CLBG benchmarks. For the most part, different interpreters show similar trends when running the same program: large usage of GC in `binarytrees`, large usage of the JIT in `fasta` and `spectralnorm`, heavy use of JIT calls in `pidigits`, and large warmup overheads in `meteor`. The main exception to this is `revcomp` where PyPy spends most of its time in the interpreter while Pycket is able to compile and use JIT traces quickly.

Three different parts of a meta-tracing JIT VM execute the application code: the interpreter, the meta-interpreter as it records a trace of the interpreter, and the JIT-compiled code. However, the performance between these different components can vary significantly. The cross-layer characterization methodology allows us to precisely determine the bytecode execution rates across these three different parts. Figure 6.4 shows the rate for the three components in terms of bytecodes executed per dynamic x86 instruction. Unsurprisingly, JIT-compiled code is typically more than two orders of magnitude faster than running the same code on the interpreter. However, as Figure 6.4(a) shows, there is a very large variation in the JIT performance (notice the log scale). Furthermore, the best performing benchmarks typically have higher performance in the JIT-compiled regions. In comparison, the interpreter and meta-interpreter performance is relatively similar across the benchmarks. Another important observation is that meta-interpreter performs roughly two orders of magnitude slower than the interpreter. While compilation passes are relatively cheap in meta-tracing JITs due to linear traces, the meta-tracing part itself can be a bottleneck.

6.5.3 Framework-Level Characterization: JIT Calls

Figures 6.2 and 6.5 show that for many benchmarks, the framework is mostly in the *JIT call phase*. These calls to AOT-compiled functions typically happen at a very fine granularity, unlike our

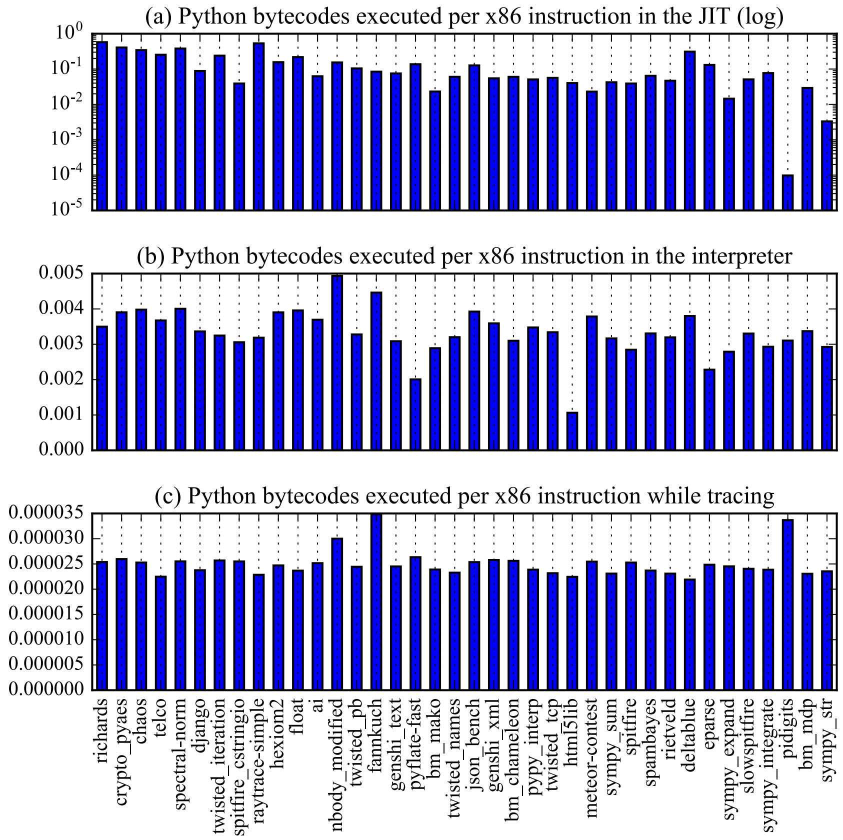


Figure 6.4: Python Bytecode Execution Rates – Bytecodes executed per dynamic x86 instruction in the (a) JIT-compiled code, (b) interpreter, and (c) meta-interpreter.

cross-layer methodology, many other measurement methodologies typically lump this phase into the JIT phase. These calls arise from functions in the interpreter or the meta-tracing framework that cannot be inlined into the trace (typically because they contain unbounded loops).

To determine which AOT-compiled functions are being called from JIT-compiled meta-traces, we tracked the target addresses when our PinTool observed a call from JIT-compiled code. Table 6.3 shows the functions that constitute at least 10% of the *overall* execution time. Note that if these functions call other functions, the time spent in the called functions is also counted as part of these entry points. Some of the functions are part of RPython-level type intrinsics which implement the operations over standard Python types (e.g., lists, strings, dictionaries) used in the interpreter and framework. Another source is the RPython standard library, which provides a subset of Python's standard library for use by the interpreter and framework. There are also external C functions usually part of the C standard library. In addition, some of these functions are defined by the interpreter or Python modules. In particular we can see that many benchmarks spend a significant amount of

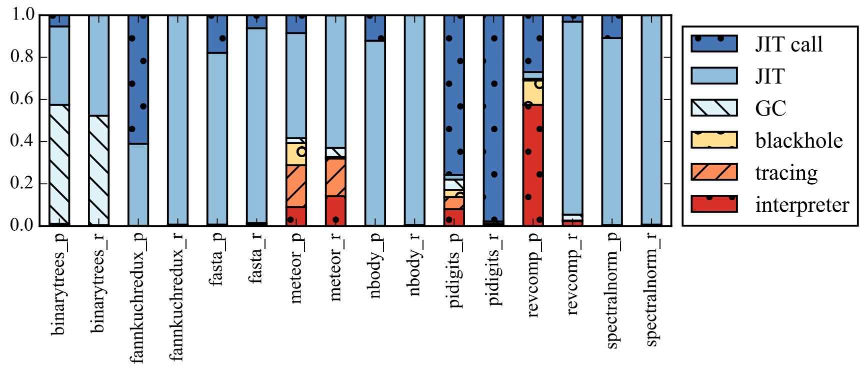


Figure 6.5: RPython Framework Breakdown – Shows the breakdown of time spent in various components of the RPython framework in CLBG. The PyPy and Pycket implementations are suffixed with `_p` and `_r` respectively.

time in `rordereddict.ll_call_lookup_function`, which is the hashmap lookup function of RPython's dictionary data structure.

6.5.4 Interpreter-Level Characterization: JIT Warmup

Warmup can have important performance implications: compiling traces too eagerly results in wasted work, and compiling traces too lazily results in wasted opportunity. Traditional characterization methods can struggle to capture detailed warmup behavior. This is because the overhead associated with the process of measuring might alter the measured performance. We insert cross-layer annotations at the interpreter level at the beginning of each iteration of the dispatch loop. This enables accurately measuring the bytecode execution rate using a PinTool, and enables a precise definition of completed work per unit time. This data can enable finding the break-even points where JIT-compiling can compile efficient code that amortizes the overhead of tracing and compiling.

Figure 6.6 shows the warmup curves of the benchmarks, normalized to CPython. These plots show the number of bytecodes executed per unit time compared to CPython over the first 10 billion assembly instructions executed. It also shows the PyPy warmup break-even points for the point in time where the number of bytecodes executed thus far on PyPy matches that on CPython (dashed vertical lines) and PyPy without JIT (dotted vertical lines). In PyPy, it is surprising that the meta-tracing JIT compilation incurs negligible slowdowns compared to running the code on a PyPy interpreter without a meta-tracing JIT. The break-even point for PyPy compared to PyPy without JIT is usually reached very early on in the programs. There is more variability for the break-even points of reaching CPython performance. The programs where PyPy's performance advantage is smaller

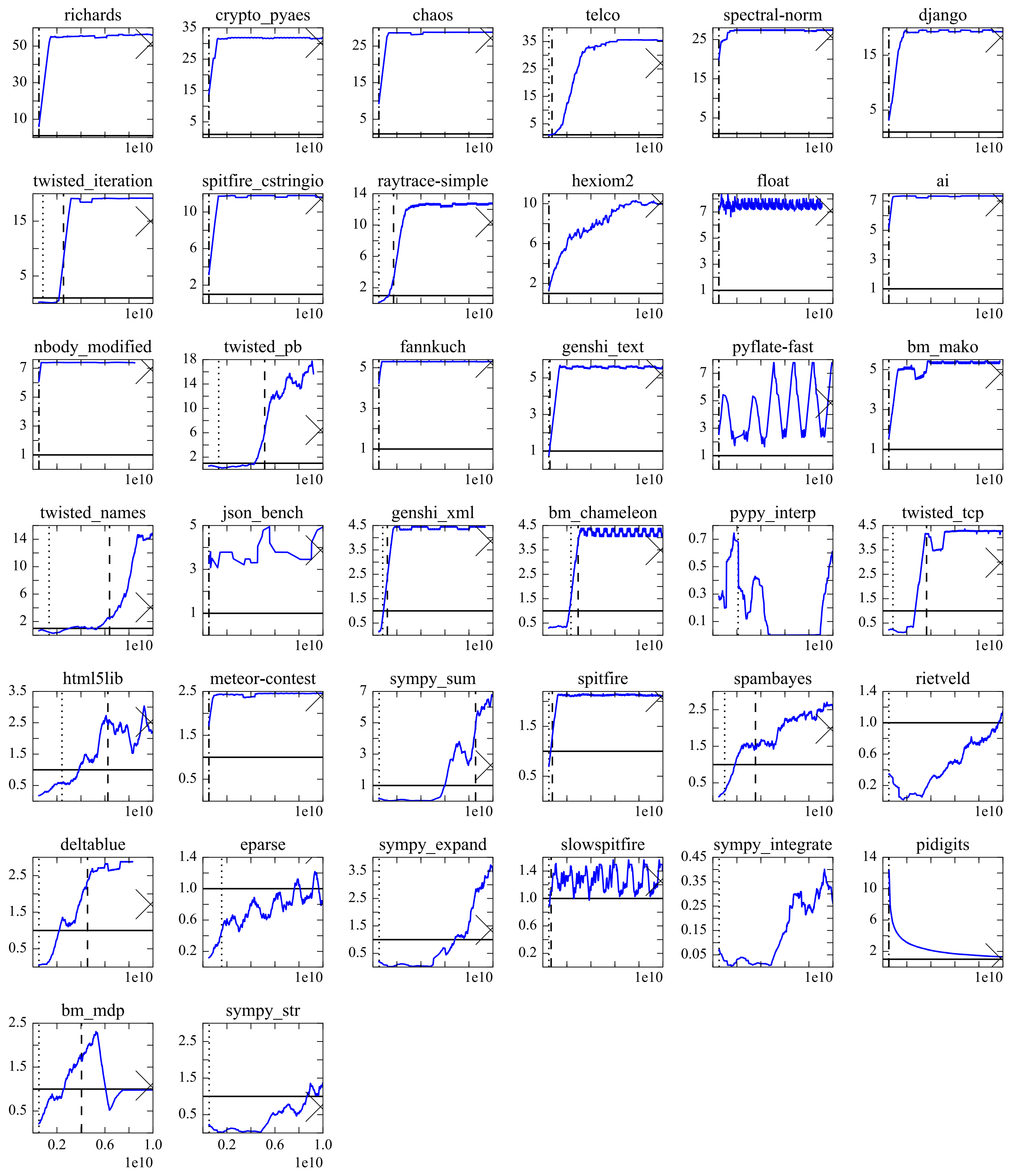


Figure 6.6: PyPy Warmup – PyPy bytecode execution rate normalized to CPython for the first 10 billion instructions. The dashed vertical lines indicate the break-even point with respect to CPython: in this point in time, both PyPy and CPython have executed the same number of bytecodes. The dotted vertical lines indicate the break-even point with respect to PyPy without JIT. The cross indicates the eventual (at the end of the execution) speedup of PyPy compared to CPython. The benchmarks are sorted in the order of speedup over CPython.

Benchmark	%	Src	Function
ai	19.4	I	setobject.get_storage_from_list
bm_chameleon	17.9	R	rordereddict.ll_call_lookup_function
bm_mako	26.1	L	runicode.unicode_encode_ucs1_helper
bm_mako	12.8	R	rordereddict.ll_call_lookup_function
bm_mdp	16.8	R	rordereddict.ll_call_lookup_function
django	16.6	L	rstring.replace
django	14.8	R	rordereddict.ll_call_lookup_function
eparse	12.3	R	rstr.ll_join
fannkuch	20.0	I	IntegerListStrategy._setslice
fannkuch	15.9	I	IntegerListStrategy._fill_in_with_sliced...
genshi_xml	12.4	R	rordereddict.ll_call_lookup_function
hexiom2	10.8	I	IntegerListStrategy._safe_find
html5lib	13.1	I	W_UnicodeObject._descr_translate
json_bench	18.5	M	_pypyjson.raw_encode_basestring_ascii
json_bench	10.6	R	rbuilder.ll_append
meteor-contest	35.4	I	BytesSetStrategy.difference_unwrapped
meteor-contest	22.2	I	BytesSetStrategy.issubset_unwrapped
nbody_modified	44.6	C	pow
pidigits	36.1	L	rbigint.divmod
pidigits	33.2	L	rbigint.int_mul
pidigits	13.0	L	rbigint.lshift
pidigits	12.6	L	rbigint.add
pyflate-fast	16.1	R	rstr.ll_find_char
pyflate-fast	11.7	I	BytesListStrategy.setslice
spitfire	22.1	R	rstr.ll_join
spitfire	14.4	R	rstr._ll_strhash
spitfire_cstringio	14.6	R	rbuilder.ll_append
spitfire_cstringio	14.1	R	ll_str.ll_int2dec
telco	13.4	L	rarithmetic.string_to_int
twisted_tcp	16.6	C	memcpy

Table 6.3: Significant AOT-Compiled Functions From Meta-Traces – Significant (>10% of overall execution) functions. The percentages are the time spent in AOT-compiled functions in overall execution. Src is where the functions are defined: R = RPython type system intinsics; L = RPython's std lib; C = external stdlib call; I = interpeter; M = PyPy module.

tends to have break-even points that are later. These benchmarks tend to be more complicated and have many different traces, so the warmup tends to take longer.

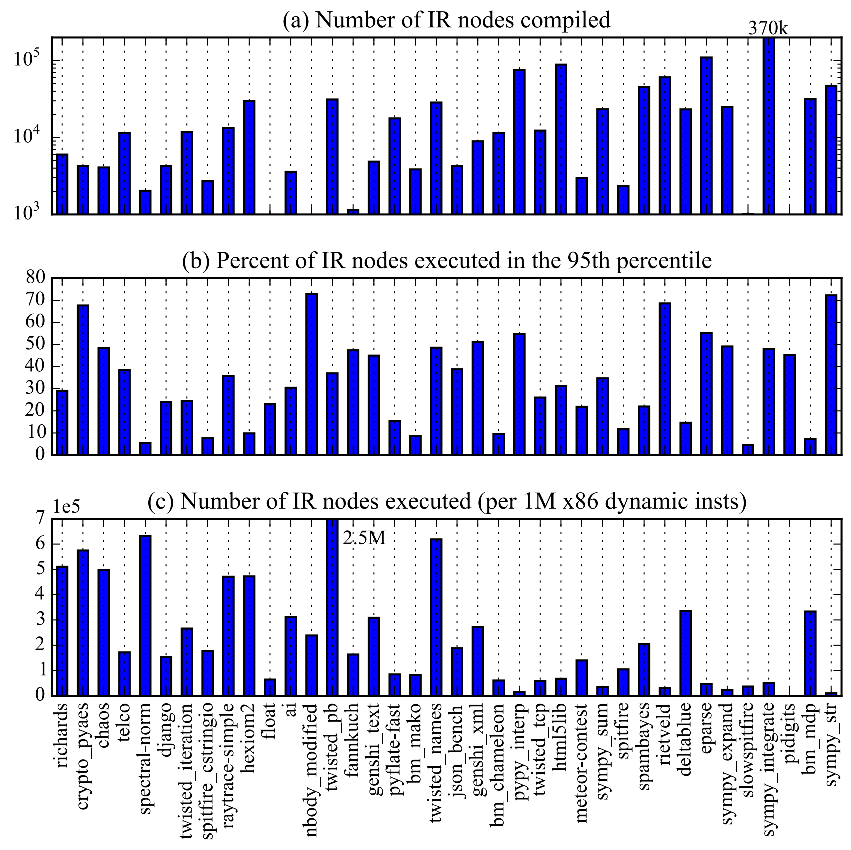


Figure 6.7: JIT IR Node Compilation and Execution Statistics – (a) Total number of JIT IR Nodes compiled throughout the benchmarks (every benchmark executed for 10B instructions), shown in log scale. (b) Most commonly executed JIT IR nodes (95% of the time spent in JIT-compiled code) shown as percentage of all IR nodes compiled. (c) Total (dynamic) number of IR nodes executed for every one million of assembly instructions executed. The benchmarks are sorted in the order of speedups over CPython.

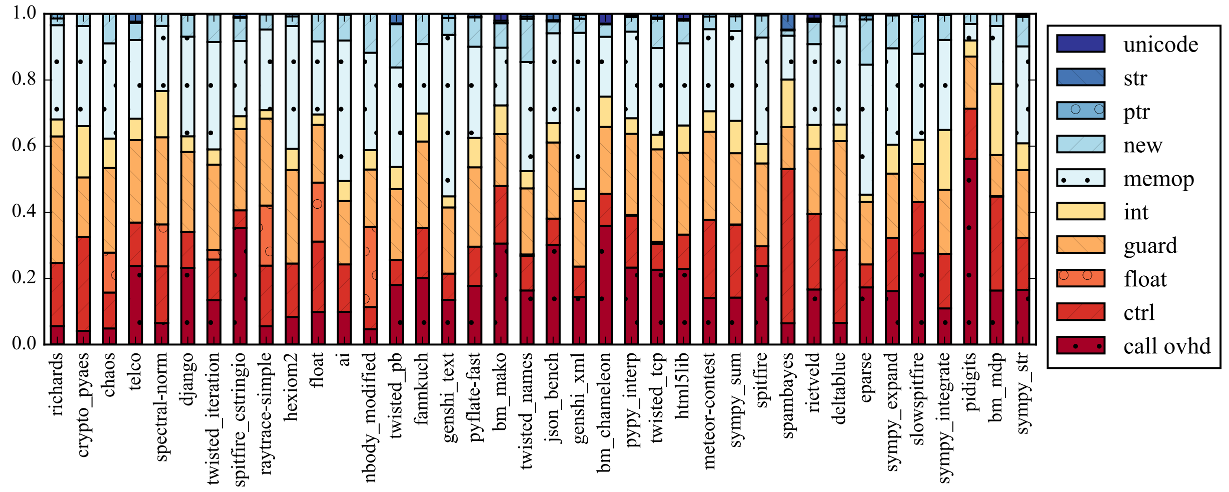


Figure 6.8: Categorized Weighted IR Node Frequency by Benchmark – Shows categorized breakdown of roughly the ratio of time spend in each class of IR nodes.

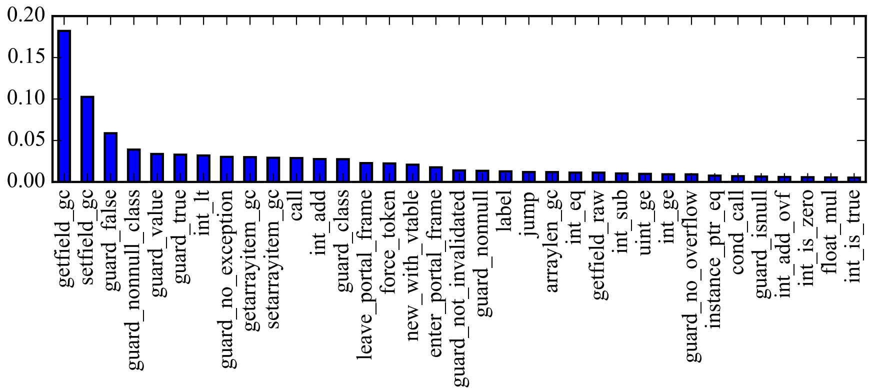


Figure 6.9: IR Node Frequency – In the PyPy Benchmark Suite, shows the frequency of the most common 35 IR node types.

6.5.5 JIT IR-Level Characterization: Compilation Burden and Usefulness of Meta-Traces

One potential drawback of tracing-based JITs as opposed to method-based JITs is long and redundant traces. Whereas the unit of compilation in a method-based JIT is typically an application-level method, for tracing-based JITs, it is a particular trace taken as an application-level loop is executed. Different functions called as the loop executes are inlined into the trace, and different control paths taken result in different traces. Tracing-based JITs therefore tend to perform poorly when the application-level code has many alternative control flow paths that are taken in similar probabilities. Such code can result in the compilation of many traces, most of them infrequently used. JIT compilation of unused traces can be a compilation burden and hurt the performance, especially for long traces. Loops with long bodies (or loops that call many functions) result in long traces. The main drawback of long and infrequently used traces is the time it takes to compile them may not be amortized especially since some compiler passes have superlinear complexities with respect to the size of the code. The secondary drawback is increased memory usage to store the generated code.

Figure 6.7(a) shows the number of IR nodes that are JIT-compiled, which shows a large variability, ranging from less than 1000 (`float`, `nbody_modified`, `slowspitfire`, and `pidigits`) to 370,000 (`sympy_integrate`). Compiling a small number of IR nodes can indicate either the program does not have many branches due to its arithmetically heavy nature, or in the case of `pidigits`, spends most of its time in calls to AOT-compiled functions for arithmetic operations. While there is a large variability across different benchmarks, the best performing benchmarks typically compile between 2000-20,000 IR nodes. Figure 6.7(b) shows the percentage of JIT IR nodes that are executed 95% of the time spent in the JIT. For some benchmarks (e.g., `spectral-norm`, `spitfire_cstringio`, `slowspitfire`, and `bm_mdp`), only 5% of the compiled IR nodes are executed 95% of the time, indicating these benchmarks have exceptionally "hot" regions within the JIT-compiled code. For these benchmarks, multi-tiered JIT compilation might be beneficial even though this is not currently supported in RPython. If a large number of JIT-compiled IR nodes are used in the 95th percentile, this shows that many traces are used equally (e.g., in `sympy_integrate` and `sympy_str`) indicating a very branchy application and large compiler burden (as Figure 6.2 also shows large tracing and blackhole overheads). Figure 6.7(c) shows the number of executed IR nodes per one million dynamic assembly instructions. This data mostly matches the time spent in JIT-compiled code in Figure 6.2. The variations are mostly due to some IR nodes mapping to different number of x86 assembly instructions. It can also be seen that the best performing benchmarks on PyPy tend to have higher than average dynamic number of executed IR nodes.

6.5.6 JIT-IR-Level Characterization: Composition of Meta-Traces

Figure 6.9 shows the *dynamic* frequency histogram of different IR node types encountered in the PyPy Benchmark Suite, based on how many times these nodes are executed. Interestingly, 80% of IR node types constitute less than 1% of the overall execution in the JIT-compiled traces (mostly for uncommon use cases). Two IR nodes, `getfield_gc` and `setfield_gc`, constitute more than 18% and 10% respectively of all JIT traces. Implied by their name, these two operations get or set a field from a pointer, resulting in a memory load or a store after pointer arithmetic.

However, the frequency alone of these IR node types does not indicate how expensive they are. For this, Figure 6.10 shows on average, how many x86 assembly instructions are required to implement each IR node type. We can see that the top IR node type is `call_assembler` which maps to more than 30 assembly instructions. Other types of calls also take up more than 15 assembly instructions. The `call_assembler` calls another JIT trace from this trace, while the other `call_` nodes call AOT-compiled functions. These values are the call *overheads*, not the time spent in the called functions. However, most IR nodes, including the common `getfield_gc` and `setfield_gc`, require only one or two assembly instructions.

Figure 6.8 shows the breakdowns with IR nodes categorized as: memory operations (memop), guards, call overheads, control flow (ctrl), integer operations (int), memory allocation (new), floating-point operations (float), string operations (str), pointer manipulations (ptr), and unicode operations.

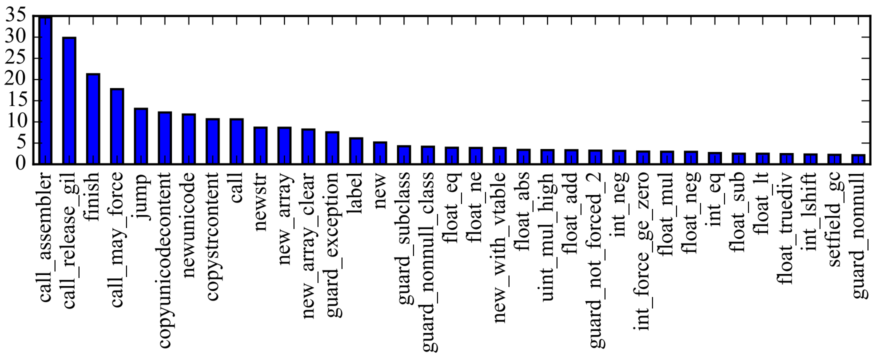


Figure 6.10: Average Number of Assembly Instructions of IR Nodes – In the PyPy Benchmark Suite, shows the average number of assembly instructions to implement the top 35 most expensive IR node types.

Across all benchmarks, memory operation IR nodes are the most common, around 26%, followed by guards at 22%, call overheads at 18%, and control flow at 16%. Memory operations are the biggest part of meta-traces, followed by guards, which are unique to JIT-compiled code. Call overheads are also a major part of meta-traces.

Looking at how these categories break down per benchmark, we see that memory operations are the most significant part of meta-traces for most benchmarks, which likely is due to Python code that makes it easy to work with complex data structures. Exceptions to this are `bm_chameleon`, `bm_mako`, `bm_mdp`, `fannkuch`, `pidigits`, `spambayes`, `spectralnorm`. We see that `bm_chameleon`, `pidigits`, and `spitfire_cstringio` have large call overheads, as these benchmarks cause many calls to AOT-compiled functions. As an example, `pidigits` is making very heavy usage of `bignum` arithmetic, which is all implemented in AOT-compiled code that the meta-traces call into. We see that guard percentages stay similar across different benchmarks except for `richards` where it constitutes most of the execution. We see that even in the arithmetic-intensive benchmarks (e.g., `float`, `nbody_modified`, `chaos`), integer and floating point operations do not constitute a significant portion of meta-traces.

6.5.7 Microarchitecture-Level Characterization

Table 6.1 shows the instructions per cycle (IPC) and branch misses per 1000 instructions (MPKI) for the benchmarks. There is a high variance (standard deviation is 0.37, 0.30, and 0.41 respectively for CPython, PyPy without JIT, and PyPy with JIT) in IPC across all benchmarks, indicating the application has a major impact on the IPC. However, CPython has a better IPC than PyPy with JIT by 15%, and PyPy without JIT by 32%. The IPC difference between the two interpreters without the JIT is surprising and partially accounts for the 2× performance gap between the two. This is again likely because RPython is not optimized for use as an interpreter without a meta-tracing JIT.

The JIT-compiled code includes numerous guards to ensure the observed control paths still hold true. This might lead to an increase in the number of branches. However, the results show that the branch rate across all interpreters is almost identical and different benchmarks do not show much variation. The MPKI of CPython and PyPy without JIT is very similar, however when the JIT is enabled, the MPKI drops by 35%. This is likely due to the more specialized and denser code produced by the JIT helping the processor to better predict the control flow [RSS15].

The cross-layer annotations also allow us to study microarchitectural characteristics by phases. Table 6.4 shows the mean and standard deviation of IPC, branches per instruction, and branch miss rate for each phase. As the overall results suggested, the interpreter IPC tends to be low with a relatively large misprediction rate. This is partially due to the interpreter being used only at the beginning for a short amount of time. The JIT phase (in this table, this also includes calls to AOT-compiled functions from the JIT-compiled code) has the largest IPC mean and the largest variation. It also has the lowest miss rate. The higher variation in the microarchitectural values is due to the application-specific nature of JIT-compiled code. The blackhole interpreter has the worst IPC among the phases, making the observations in Section 6.5.2 regarding the expense of this phase even more significant. Finally, the GC phase has a relatively high IPC compared to the other phases, perhaps because the same collection code is executed over and over, allowing the predictors to warm up sufficiently.

6.6 Discussion

In this section, we use the results from Section 6.5 to derive initial answers to the nine key questions listed in Section 6.1.

1. Can meta-tracing JITs significantly improve the performance of multiple dynamic languages? While it is well known that JIT compilation can significantly improve the performance of dynamic languages, the performance of meta-tracing JITs across different languages is still an active research area. As Table 6.1 showed, the PyPy meta-tracing JIT was indeed able to out-perform the CPython interpreter on almost all benchmarks (up to 51× on `richards` and 30× on `crpyto_pyaes`). Looking at the CLBG results in Table 6.2, the Pycket meta-tracing JIT had comparable performance to

Phase	IPC	Branch / inst	Branch miss rate
interpreter	0.76 (0.26)	0.15 (0.019)	0.06 (0.021)
tracing	1.05 (0.07)	0.15 (0.003)	0.05 (0.005)
JIT	1.24 (0.53)	0.16 (0.038)	0.02 (0.026)
blackhole	0.48 (0.10)	0.13 (0.009)	0.09 (0.019)
GC	1.18 (0.30)	0.20 (0.024)	0.04 (0.016)

Table 6.4: Microarchitectural Characterization of Phases – Microarchitectural means (and standard deviation in parentheses) by RPython phase in the PyPy Benchmark Suite.

Racket, a custom JIT-optimizing VM. Even though there are some exceptions, and certainly room for improvement, our results seem to confirm the promise of meta-tracing JITs.

2. Does generating and optimizing traces in a meta-tracing JIT add significant overhead? The conventional wisdom is that JIT warmup can add a significant overhead, which might be problematic especially when interactivity is important. We see in Figure 6.2 that tracing can consume a large percentage of the total execution time for certain benchmarks. However, Figure 6.6 shows that the break-even point where meta-tracing JIT performance exceeds PyPy without the meta-tracing JIT occurs early in the benchmark execution. This suggests that even for short running applications, enabling the meta-tracing JIT does not significantly reduce performance. So compared to the basic PyPy interpreter, the overhead is not as significant as the conventional wisdom might suggest. CPython's interpreter is faster than PyPy without the meta-tracing JIT, so the corresponding break-even point is somewhat later in the execution.

3. Does deoptimization in a meta-tracing JIT consume a significant fraction of the total execution time? The conventional wisdom is that deoptimization should be relatively inexpensive [HCU92, KM05, SB12]. Figure 6.2 shows that deoptimization (implemented using the "blackhole interpreter" in RPython) can consume more than 10% of the total execution time for some benchmarks. The phase diagrams of both fast-warming and slow-warming benchmarks in Figure 6.3 illustrate that the blackhole phase is an essential part of the warmup process. Meta-traces compiled when the control-flow coverage is insufficient require many deoptimizations to fall back to the interpreter and compile additional meta-traces. Furthermore, Table 6.4 suggests that the blackhole phase performs poorly on modern hardware. Our results suggest that deoptimization presents a more significant overhead than perhaps the conventional wisdom might suggest.

4. Does garbage collection in a meta-tracing JIT consume a significant fraction of the total execution time? While garbage collection (GC) used to be a significant worry, modern JIT-optimizing VMs are carefully constructed to ensure that GC only consumes a small fraction of the total execution time (e.g., Wilson writes it "should cost roughly ten percent of running time" [Wil92], which Jones and Lins call a "not unreasonable [figure] ... for a well-implemented system" [JL96, p. 13]). The breakdowns in Figure 6.2 confirm that RPython's GC does indeed consume a reasonable fraction of the total execution time with the exception of a few memory-intensive benchmarks.

5. Is all JIT-compiled code equally used in a meta-tracing JIT? The conventional wisdom states that there is different levels of "hotness" of frequently executed code leading to multi-tiered JITs, where the "hotter" a code region is, the more the compiler will try to optimize. However, this effect might be less pronounced in a tracing JIT due to each trace of execution getting compiled separately, so there might be many JIT-compiled traces which are executed roughly equally. Figure 6.7 shows that there is indeed a large "hotness" variability where in some benchmarks only 5% of compiled IR nodes are executed 95% of the time. For these cases, multi-tiered JIT compilation indeed should help.

6. Does a meta-tracing JIT need to spend most of its time in the JIT-compiled code to achieve good performance? Because JIT-compiled code is optimized using run-time feedback, the conventional wisdom suggests that most of the time should be spent executing JIT-compiled instructions. However, Figure 6.2 showed that many benchmarks (including several high-performance benchmarks) spend more of their time in AOT-compiled functions than JIT-compiled code. Because irregular control flow can potentially reduce performance, there is a delicate balance between code which should be JIT-compiled (to benefit from run-time feedback), and code which should be AOT-compiled (to avoid generating many bridges).

7. What fraction of the time in the JIT-compiled code is overhead due to the meta-tracing JIT? The conventional wisdom is that it is difficult to optimize dynamic languages, so overheads such as pointer chasing to access high-level data structures cannot be easily optimized away [CEI$^+$12]. If we look at the breakdown in Figure 6.8, we see that memory operations indeed constitute the most significant time of JIT-compiled code. In addition, most guards also represent a JIT-specific overhead. Finally, calling AOT-compiled functions brings in a substantial overhead. While it is hard to name an exact percentage, it is likely that more than half of the JIT-compiled code is overhead.

8. What is the microarchitectural behavior of JIT-compiled code? The conventional wisdom states that JIT-compiled code might be inefficient compared to AOT-compiled code or a pure interpreter due to additional overheads such as guards. Comparing the performance to statically compiled languages and CPython in Table 6.2, we see that meta-tracing JITs indeed have lower IPC. However, the IPC is usually within 0.5 of C code, meaning the gap is not as significant as one might assume. Furthermore, comparing the microarchitectural behavior of the RPython interpreter with and without the meta-tracing JIT in Table 6.2 and the phase microarchitectural breakdown in Table 6.4, we can see that the JIT-compiled code has better microarchitectural behavior than other phases of the RPython framework.

9. Why are meta-tracing JITs for dynamic programming languages still slower than statically compiled languages? Holistically considering our workload characterization, it is clear that there is no single reason. The meta-tracing framework has many components that different benchmarks and languages stress in different ways. The microarchitecture-level results do suggest that the primary problem is less that the JIT-compiled code is difficult to execute efficiently on modern architectures, and more that meta-tracing JITs still do significantly more work than statically compiled languages. Overall, there is no single "silver bullet" to improve meta-tracing JIT performance, meaning there is a wide array of opportunities for future researchers from both the VM and architecture communities.

6.7 Related Work

To our knowledge, this is the first multi-language, cross-layer workload characterization that focuses on meta-tracing JITs.

Sarimbekov et al. study the workload characteristics of various dynamic languages on the JVM using the CLBG benchmarks [SPB$^+$13]. They find that even though the implementations use many polymorphic callsites, most runtime method invocations are actually monomorphic. They also find that dynamic languages allocate significantly more objects than Java, most of which are short-lived, due to the additional boxing/unboxing that is common in dynamic languages. Ismail and Suh present a workload characterization of PyPy and CPython [IS18]. Their characterization focuses primarily on garbage collection and managed memory and is orthogonal to this work.

Rohou et al. study branch misprediction in switch-based interpreters on Haswell-generation Intel hardware [RSS15]. They find that counter to folklore [EG01], the cost of branch misprediction is very low on modern hardware, so techniques like jump threading are no longer necessary. However the statement of Rohou et al. that the "principal overhead of interpreters comes from the execution of the dispatch loop" is itself called into question by the work of Brunthaler [Bru09]. He argues that this "is specifically not true for the interpreter of the Python programming language." The reason he gives is that Python bytecodes do a lot of work, so the overhead of bytecode dispatch is relatively lower. Castanos et al. observe this as well [CEI$^+$12]. They write: "[in Jython] a single Python bytecode involves between 160 and 300 Java bytecodes, spans more than 20 method invocations, and performs many heap-related operations. Similar path lengths are also observed in the CPython implementation."

Anderson et al. introduce the "Checked Load" ISA extensions to offload type checking overhead to hardware in dynamic language interpreters [AFCE11]. Choi et al. propose a similar hardware mechanism for object access [CSGT17]. Gope et al. identify calls to AOT-compiled functions from the PHP JIT as significant overhead and propose hardware accelerators for common framework-level functions [GSL17]. Dot et al. present a steady-state performance analysis of the V8 JavaScript engine using the builtin sampling profiler and present hardware mechanisms to provide a 6% performance speedup [DMG15]. Southern and Renau use real systems to characterize the overhead of deoptimization checks (guards) in V8 [SR16]. Like our findings, they also find that the cost of such checks is lower than the conventional wisdom might suggest in the context of V8.

Würthinger et al. provide an overview of the Truffle system and the concept of runtime calls (AOT-compiled calls) from the JIT-compiled code, but do not quantify the overheads of such calls [WWH$^+$17].

Holkner and Harland evaluate the dynamic behaviour of Python applications [HH09]. They find that all the tested programs use some dynamic reflective features that are hard to compile statically, and a large fraction of the tested programs execute code that is dynamically generated at runtime (20%).

There have been some studies to characterize the performance of JavaScript, e.g., [RLBV10] study the dynamic features typically used in JavaScript, and [RLZ10] study the correlation between benchmarks and their real-world counterparts.

6.8 Conclusion

We have presented a cross-layer workload characterization of meta-tracing JIT VMs. To make this study possible, we first introduced a new cross-layer annotation methodology, which allows inserting annotations at different abstraction levels of a multi-level VM (e.g., the RPython framework), and observing these at different levels of execution (e.g., the binary level to get microarchitectural statistics, or the assembly layer to intercept using a dynamic binary instrumentation tool). We then used two RPython interpreters, PyPy and Pycket, for application-level, framework-level, interpreter-level, JIT IR-level, and microarchitecture-level characterization. We finally provided initial answers to nine key questions regarding meta-tracing JIT VM performance. One main takeaway is the performance characteristics are highly varied across different applications, bottlenecked by different parts of the meta-tracing JIT VM, and there is likely no single "silver bullet" that could give significant speedups by a small change in software or hardware.

CHAPTER 7
SOFTWARE/HARDWARE CO-DESIGN TO EXPLOIT OBJECT DEREFERENCE LOCALITY

As Chapter 6 shows, there are a number of diverse factors that can reduce the performance of applications that execute on meta-tracing JIT VMs. This unfortunately means that there is no "silver bullet" hardware acceleration idea that can significantly improve the performance across the board. However, the data also suggest that the time spent while executing JIT-compiled code is a significant part of the overall execution. Hardware acceleration techniques targeting the execution of JIT-compiled code can have tangible performance benefits on many benchmarks. In this chapter, I identify one promising type of value locality called object dereference locality in the JIT-compiled code. Then I propose a software/hardware co-design approach to exploit this pattern to reduce dynamic instruction count by up to 40% while executing JIT-compiled code.

7.1 Introduction

Dynamic programming languages are growing in popularity across the computing spectrum from smartphones (e.g., JavaScript for mobile web clients), to servers (e.g., Node.js, Ruby on Rails), to supercomputers (e.g., Julia for technical computing). Dynamic programming languages such as Python, JavaScript, PHP, Ruby, and MATLAB are all now among the top-ten most popular programming languages. Dynamic languages typically provide productivity-focused features such as: lightweight syntax, dynamic typing of variables, managed memory, rich standard libraries, interactive execution environments, and advanced reflection capabilities. However, simple interpreters for dynamic languages are often multiple orders-of-magnitude slower than statically compiled languages such as C/C++. While dynamic languages are not amenable to ahead-of-time compilation, just-in-time-optimizing virtual machines (JIT VMs) [Ayc03, AFG$^+$05] can improve performance by identifying frequently executed code paths and the concrete types of variables in these locations, and generating type-specialized machine code for these hot paths in a just-in-time fashion [BCFR09, BT15, WWW$^+$13, WWS$^+$12, Höl13, GES$^+$09, spi, jul, v8]. While state-of-the-art JIT VMs close a major part of the performance gap, it is still common to have up to an order-of-magnitude slowdown compared to C/C++. Another drawback of JIT VMs is the immense engineering effort required to build them. However, partial-evaluation-based [WWH+17, WWW+13, WWS+12] and meta-tracing-based [BCFR09, BT15] frameworks, which allow automatically building JIT VMs from dynamic language interpreters, have recently been gaining traction. VMs built using these frameworks can rival the performance of custom-built JIT VMs, and improvements to the common runtime ultimately benefit all of the dynamic language interpreters that use that runtime.

Dynamic languages allow usage patterns that do not exist in static languages: a variable can point to values of different types; types can be mutated; and reflection allows programmatically manipulating objects and classes. Because of this, VMs cannot make a priori assumptions regarding types, resulting in many indirect lookups. For example, to get a field of an object, the VM must first get the type object, then get the object map (similar to virtual method tables) from the type object, then perform a lookup in the object map to find the field slot corresponding to the field name, and finally get the field value. The JIT removes dereferences and new object allocations when it can prove it is safe to do so, but there are many cases where potential aliasing or the interaction of non-JIT-compiled code prevent such optimizations [SWM14, BCF+11a]. Chapter 6 shows that at the VM level, the execution time is mostly spent in the JIT-compiled code, but roughly half of the JIT-compiled operations are memory operations and checks, even after the safe dereferences, checks, and allocations are removed. In this chapter, we study object dereferences in the JIT-compiled code in meta-tracing JIT VMs and find that a large percentage have load-value locality: these instructions load from a small set of addresses, and the values loaded are constant with respect to the address. Additionally, we observe that at the intermediate-representation- (IR) level, these usually constant loaded values are often used by pure IR operations. These two observations form the foundation for what we call “object dereference locality”. We propose a software/hardware co-design technique to identify and exploit skippable sub-traces (SSTs) out of the larger JIT-compiled trace that have object dereference locality. We use SST profiling to identify loads with value locality in JIT-compiled traces, SST recompiling to recompile these traces and form the skippable sub-traces, SST execution to skip over the sub-traces if the hardware can prove it is safe to do so, and SST invalidation to monitor stores that might make a particular sub-trace no longer safe to skip.

The contributions of this chapter are: (1) we characterize object dereference locality in the context of meta-tracing JIT VMs; (2) we propose using software/hardware co-design to discover object dereference locality in JIT-compiled traces, recompile JIT-compiled traces to include skippable sub-traces using an ISA extension, a hardware scheme to skip these sub-traces if safe, and

```
1 sum = 0
2 for node in nodes:
3   sum += node.val
```

Figure 7.1: Python Accumulate Example

```
1  while True:
2   bc = bcs[i]
3   if bc.type == ADD:
4     v1 = stack.pop()
5     v2 = stack.pop()
6     if (v1.type == IntObj &
7         v2.type == IntObj):
8       i1 = v1.intval
9       i2 = v2.intval
10      res = add_ovf(i1, i2)
11      stack.push(res)
12    elif ...
13  elif bc.type == ...
```

Figure 7.2: Simplified Python Interpreter Snippet

```
1  label(j, n, nodes, sum)
2  i0 = (j >= n)
3  guard_false(i0)
4  p1 = nodes[j]
5  i2 = j + 1
6  iterator.index = i2
7  guard_nonnull_class(p1, UserInstanceObj)
8  p2 = p1.map
9  guard_value(p2, 9453327)
10 p3 = p1.value0
11 guard_nonnull_class(p3, IntObj)
12 i4 = p3.intval
13 i5 = sum + i4
14 guard_no_overflow()
15 jump(i2, n, nodes, i5)
```

Figure 7.3: Pseudo-IR of JIT-Compiled Trace

a hardware mechanism to detect conditions that would make it unsafe to skip the sub-traces; and (3) we evaluate the sub-trace skipping design using dynamic instruction counts and cycle-level simulation for two different meta-tracing JIT VMs for the Python and Racket languages.

7.2 Motivation

Figure 7.1 shows a simple loop from an example Python application that increments a local variable `sum` with the `val` fields of a list of `Node` objects. The interpreter would look similar to the simplified example in Figure 7.2 with an interpreter loop that performs the semantics of the current bytecode. Lines 3–11 show a subset of addition semantics in Python. Notably, the type check and integer overflow checks are required due to these semantics. Figure 7.3 shows the pseudo-IR of the JIT-optimized trace that corresponds to the application-level loop, generated through meta-tracing multiple iterations of the interpreter, until the application-level loop is completed. Since the trace was generated by executing the interpreter, we can see the equivalence of type and overflow checks on lines 7 and 10 in Figure 7.2 with lines 11 and 14 in Figure 7.3 respectively.

An important observation in the trace in Figure 7.3 is that a similar C++ code would only need to perform the operations in lines 1–5, 12, 13, and 15. The storage of the current induction variable to the iterator object in line 6 is due to higher level iterators in Python, line 7 checks that `p1` (`node`) is indeed a user-defined class (as opposed to a built-in type), and lines 8–9 check if `node` has the same map as the particular user-defined class object that was observed when the trace was built. Objects in dynamic languages are name bound: the semantics require field lookups to be equivalent to be a string-indexed lookup in a mutable dictionary (e.g., Python's `__dict__`). To reduce the number of string-indexed lookups, interpreter-level immutable data structures called maps are used to assign slots to fields of known user-defined classes [BCF+11b]. The check in line 9 thus allows skipping the string-indexed lookup into the map as long as the object has the same map, meaning the type of `node` is what was previously observed (`Node`). By this point, we know the type of `p1`, so we can type specialize: line 10 gets the `val` field using the recorded slot (`value0`), line 11 checks if `val` is an `IntObj`, line 12 unboxes the `IntObj` object to get the raw integer value, and line 14 ensures the addition that was performed did not overflow. It is important to note that while in this case the guards in lines 7, 9, and 11 will not fail, for correctness they are required in case `nodes` contains objects of a different type, the `val` field is not an `int`, or the `Node` class itself is mutated.

Allocation and dereference removal JIT optimization techniques tackle overheads such as these. The state-of-the art in these optimizations in the context of tracing JITs work at the granularity of traces [SWM14, BCF+11a]. These algorithms identify objects that are repeatedly accessed in the same trace. Then, as long as the optimizer can guarantee that there are no operations in the trace that would change the outcome of the loads, these loads are hoisted to the beginning of the JIT-compiled code. Similarly, stores and new object allocations can be moved to the exits points of the trace, allowing reuse within the trace. This optimization is able to remove on average 89% of load, store, and new allocation operations across many benchmarks [BCF+11a]. For instance, because of the allocation and dereference removal optimization, the type check, unboxing of the raw integer, and boxing the raw integer back into an `int` object for `sum` were successfully optimized away in Figure 7.3. While this is very impressive, the optimizer has to be conservative when there is any possibility of a store that can modify the outcome of a hoisted load. Furthermore, these loads still have to be performed every time a different JIT trace is entered, which reduces the performance of control-flow-intensive code.

To characterize the overhead of object dereferences that almost always return the same data (e.g., lines 8 and 10) and checks that never fail (e.g., lines 7, 9, and 11), we identify object dereferences with object dereference locality (ODL) at the IR level. These object dereferences have high load-value locality: they load from a few effective addresses and load the same data from these addresses.

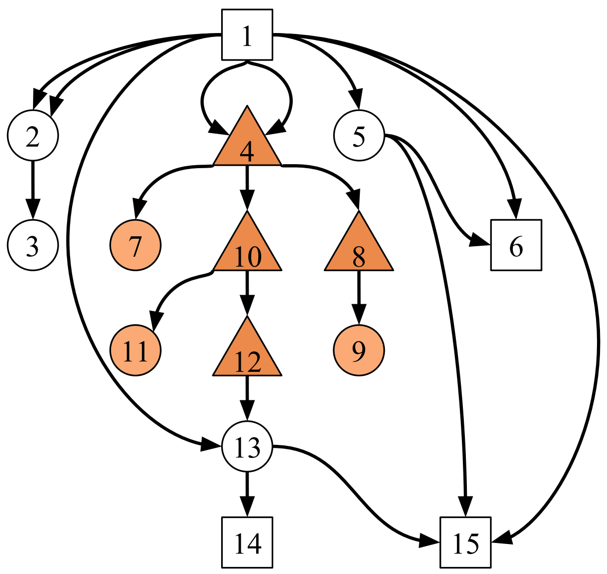


Figure 7.4: JIT IR Dataflow Graph for Accumulate Example – The numbers correspond to line numbers in Figure 7.3. Triangles are load IR nodes, circles are pure IR nodes, and squares are remaining IR nodes. Shaded triangles and circles are IR nodes with object dereference locality and derived value locality respectively.

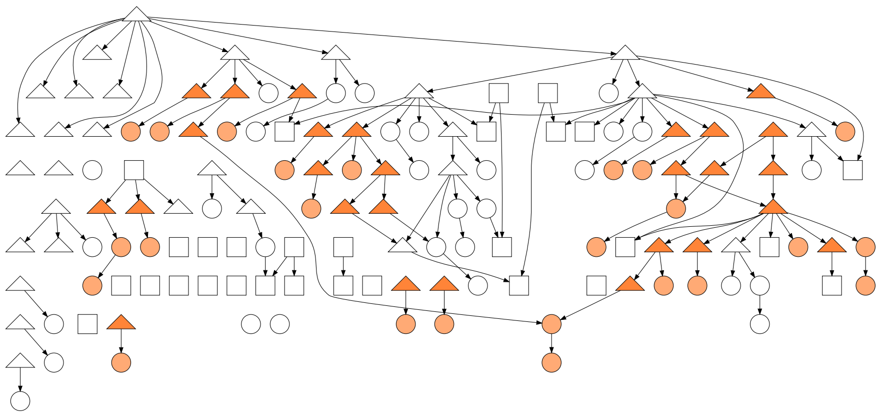


Figure 7.5: Example JIT IR Dataflow Graph from Richards

We further propagate ODL to dependent pure operations at the IR level (given the same input, these always return the same output). We call this type of locality dependent value locality (DVL). Figure 7.4 shows the dataflow graph (DFG) that corresponds to the trace in Figure 7.3. The numbers in Figure 7.4 correspond to the line numbers in the trace, and we represent IR nodes in three different shapes: triangles for object dereferences, circles for pure operations (includes most guards), and squares for the remaining IR nodes. We show the nodes with ODL and DVL using shaded triangles and circles respectively. Figure 7.5 shows a similar DFG from the `richards` benchmark. Table 7.1 shows the percentage of dynamic instructions in the traces that have ODL, DVL, and the combined

Benchmark	ODL %	DVL %	IVL %
ai	6.9	8.9	15.8
nbody	8.5	16.2	24.7
binarytree	5.7	7.9	13.6
fasta	9.1	9.3	18.4
richards	18.2	28.6	46.8
spectralnorm	7.1	9.2	16.3

Table 7.1: Value Locality in JIT-Compiled Code – Value locality measured as a percentage of the dynamic instruction count. ODL = object dereference locality; DVL = derived value locality; IVL = IR-level value locality (ODL + DVL).

IR-level value locality (IVL) for some of the Python benchmarks running on PyPy. While ODL itself is modest (5–18%), the upper bound on instruction savings in the JIT-compiled region is 13% to 46%. This potential overhead makes the case for a software/hardware co-design technique that can skip over these IR nodes that have high IR-level value locality.

7.3 Skippable Sub-Traces

To exploit IR-level value locality in meta-tracing JIT VMs, we propose a software/hardware co-design approach called *skippable sub-traces* (SST). The rest of this section explains each of the components in detail.

7.3.1 SST Profiling

In order to exploit IVL in the JIT-compiled traces, the JIT compiler has to know which operations are likely to exhibit ODL. One approach is to use software-based profiling: instructions can be inserted into JIT-compiled code to check the addresses and loaded values of object dereferences. However, this approach can potentially introduce large overheads, since more than a quarter of all operations in JIT-compiled traces are memory operations (see Chapter 6). Alternatively, interpreter-level heuristics could be used, since certain interpreter-level variables are unlikely to change. For example, line 8 in Figure 7.3 is not likely to change throughout the execution because types in programs are generally stable. However, interpreter-level heuristics would require annotating every single interpreter with these hints in order to exploit ODL. In addition, this approach would likely introduce both false positives and miss some of the opportunities. We instead propose using an

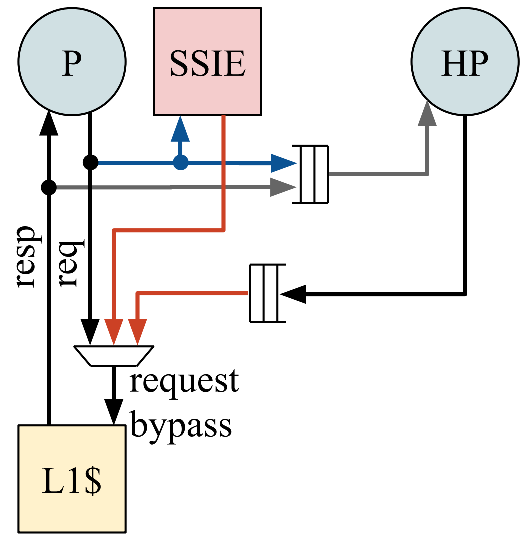


Figure 7.6: SST Architecture – Helper processor (HP) is attached to the main processor (P) using memory request/response snooping network. SSIE snoops write requests from the processor. Both SSIE and HP have special bypass channels into the L1 cache request port.

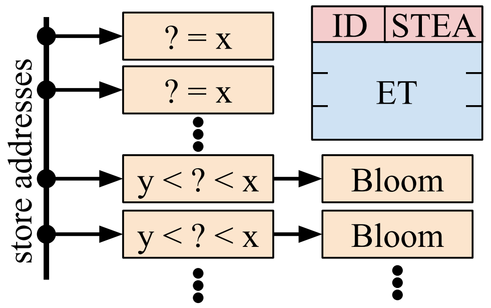


Figure 7.7: SST Synchronous Invalidation Engine (SSIE) – The entry table (ET) contains the equality checker or bloom filter ID and the corresponding sub-trace entry address (STEA) into the memory-mapped sub-trace entry region.

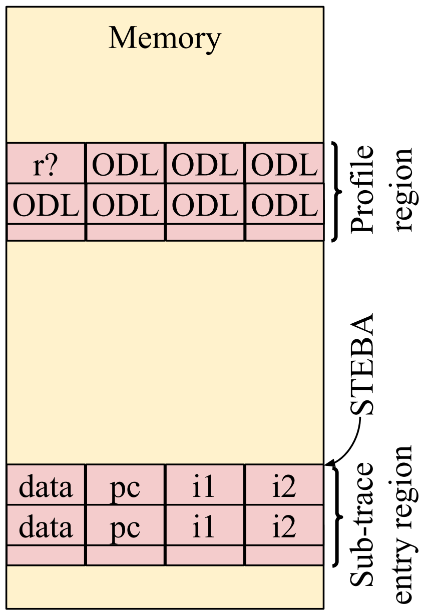


Figure 7.8: Memory Regions Allocated in Main Memory – The VM allocates a per-trace profile region and a global sub-trace entry region. The profile region contains a bit that triggers a recompilation, and the remaining bits indicate whether a given object dereference has locality. The sub-trace entry region contains one entry per sub-trace entry, with a word for the memoized data, the PC of corresponding SST instruction, and two input values. STEBA = Sub-Trace Entry Base Address.

additional processor core to identify ODL in an asynchronous fashion. Figure 7.6 shows the main processor (P) that executes the meta-tracing JIT VM, and a helper processor (HP) that is connected with a memory request/response snooping network. The JIT compiler emits code that uses specially marked load instructions for object dereference operations. The processor sets a special bit in the L1 cache request bus when it executes these marked load instructions. The memory requests with the bit set will be snooped and stored in an HP input queue. The loaded values from the L1 cache will also be matched against the request addresses in the HP input queue. Using the address and value pairs, software running on the HP can detect which loads exhibit object dereference locality.

The VM allocates an area in the virtual address space called the profile region for every JIT trace and communicates this to the HP, as shown in Figure 7.8. This enables the helper processor to asynchronously notify the JIT runtime running on the main processor with a list of object dereferences with ODL. The per-trace profile region consists of a per-trace recompile bit and ODL bits for every JIT IR-level dereference operation. The VM inserts a guard to each JIT-compiled trace to read and check if this recompile bit has been set. If the bit is set, the guard triggers recompilation routines in the VM. A set ODL bit is used to inform the VM if the indicated dereference operation has locality. Figure 7.6 also shows the memory request bypass channel into the main processor's L1 cache. This bypass channel prioritizes accesses from the HP. Once the HP reaches the threshold for required number of samples, it marks which loads have locality in the profile region, using the bypass channel. Because updates to the profile region are rare and small, and because the main processor does not read this data until a recompilation request arrives, there will relatively little memory contention using this bypass design. Once the ODL nodes have been marked, the HP will also flip the recompile bit. This flipped bit will cause a guard failure in the trace and will force a JIT recompilation of this JIT trace.

7.3.2 SST Recompilation

Once significant ODL has been identified, the JIT compiler recompiles the trace. The main goal is to start with the ODL nodes and find dependent pure operations that can be grouped together. The grouped IVL nodes are called skippable sub-traces (SSTs). Figure 7.10(a) shows the trace from Figure 7.3 after it is recompiled. Figure 7.9 shows the SSTs on the dataflow graph from Figure 7.4. Each SST has one or more inputs, zero or more outputs, and a body that contains the semantics of the sub-trace. SSTs act like pure functions as long as there are no writes that change the outcome of the dereferences inside the SSTs. Because SSTs are usually pure, their output can be memoized, and execution can skip over these pure operations. If the particular inputs to an SST have been encountered before, and if the SST has not been invalidated, then the SST body can be skipped over. Figure 7.10(b) shows the SSTs that have been formed. All of these SSTs start with an object dereference that the helper processor identified to have value locality. From these dereferences,

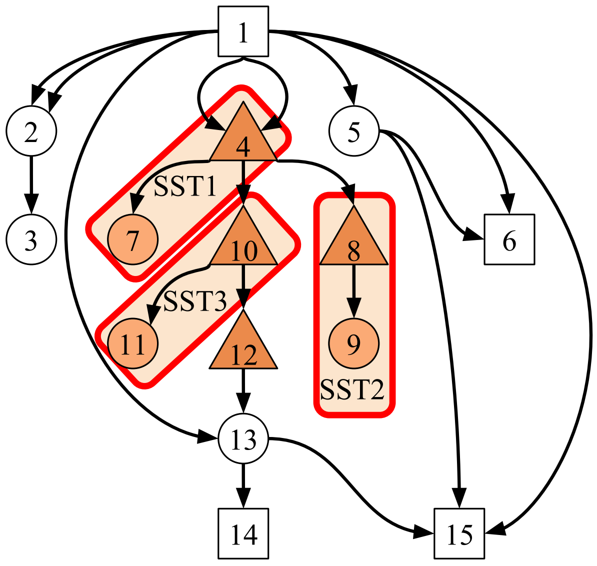


Figure 7.9: JIT IR Dataflow Graph with Conservative SSTs.

```
 1 label(j, n, nodes, sum)
 2 i0 = (j >= n)
 3 guard_false(i0)
 4 i2 = j + 1
 5 iterator.index = i2
 6 p1 = SST1(nodes, j)
 7 SST2(p1)
 8 p3 = SST3(p1)
 9 i4 = p3.intval
10 i5 = sum + i4
11 guard_no_overflow()
12 jump(i2, n, nodes, i5)
```

(a)

```
SST1(nodes, j):
 p1 = nodes[j]
 guard_nonnull_class(p1, UserInstanceObj)
 return p1
SST2(p1):
 p2 = p1.map
 guard_value(p2, 9453327)
SST3(p1):
 p3 = p1.value0
 guard_nonnull_class(p3, IntObj)
 return p3
```

(b)

Figure 7.10: Conservative SSTs – Recompiled Traces from Figure 7.3 that correspond to the dataflow graph in Figure 7.9. (a) Trace recompiled with skippable sub-traces (SSTs). (b) SST operations.

subsequent dependent pure operations have been added to the SSTs to grow them. The returned values can be used by the rest of the trace and potentially used as inputs to other SSTs (e.g., `p1`).

Note that the SSTs formed in Figure 7.10 are conservative. Line 9 in Figure 7.10(a) is not part of any SST, even though this dereference has locality. The reason for this is because `i4` does not have a pure dependent operation (even though line 10 is pure, it has an additional input, `sum`, that does not have value locality). We do not allow single-node SST bodies because there would not be any benefit to skipping the sub-trace. Checking if a sub-trace can be skipped costs at least one

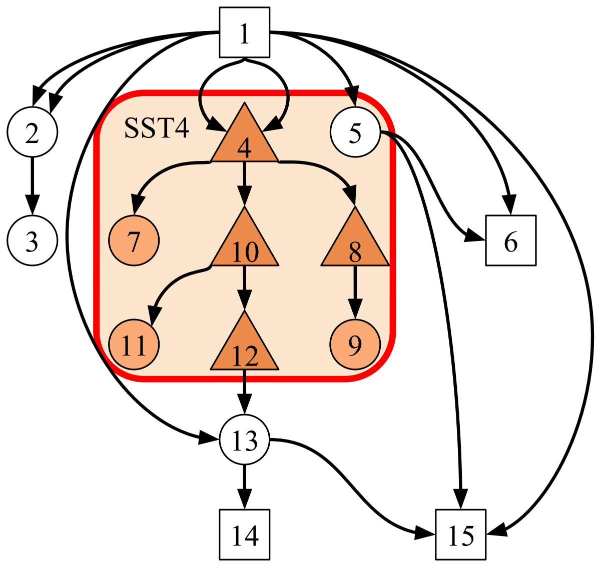


Figure 7.11: JIT IR Dataflow Graph with Aggressive SSTs.

```
label(j, n, nodes, sum)
i0 = (j >= n)
guard_false(i0)
i4, i2 = SST4(nodes, j)
iterator.index = i2
i5 = sum + i4
guard_no_overflow()
jump(i2, n, nodes, i5)
```

(a)

```
1  SST4(nodes, j):
2    p1 = nodes[j]
3    i2 = j + 1
4    guard_nonnull_class(p1, UserInstanceObj)
5    p2 = p1.map
6    guard_value(p2, 9453327)
7    p3 = p1.value0
8    guard_nonnull_class(p3, IntObj)
9    i4 = p3.intval
10   return i4, i2
```

(b)

Figure 7.12: Aggressive SSTs – Recompiled Traces from Figure 7.3 that correspond to the dataflow graph in Figure 7.11. (a) Trace recompiled with skippable sub-traces (SSTs). (b) SST operations.

instruction. Furthermore, SST2 and SST3 are dependent on the output of SST1, so it is possible to grow them to a single larger SST.

Figure 7.12 shows the trace after using an algorithm that grows the SSTs much more aggressively. Figure 7.11 shows the aggressive SST on the dataflow graph. In this particular case, all of the overhead operations can be reduced to a single large SST by chaining the nodes with ODL and subsequent dependent operations. Also note that non-dependent operations that use the inputs of the SST can be added (e.g., line 3 in Figure 7.12(b)). Growing SSTs also removes the need to return intermediate values altogether (e.g., p1, p2, and p3). Aggressive SSTs can enable higher performance by skipping over more code. However, aggressive SSTs also contain more object dereferences. Even though these dereferences have been identified to contain locality during profiling, they are not guaranteed to exhibit the same behavior for the remainder of the execution. A store executed elsewhere can alter the values that the dereference loads. This requires invalidating the SST so that the values can be computed again with the new dereference values. Since aggressive SSTs contain more dereferences, the likelihood of any one of them causing an invalidation is significantly higher than a conservative SST.

7.3.3 SST Execution

Executing skippable sub-traces requires two techniques: SST entries and ISA extensions. SST entries may either be stored in a dedicated scratchpad or in the memory system. If stored in the memory system, similar to the profile region, the VM also allocates a page of memory in the virtual address space called the sub-trace entry region (see Figure 7.8). This region consists of cache-line-aligned sub-trace entries, where each of the entries contains the sub-trace PC, input values, and the memoized data associated with the entry. Every unique input combination to an SST will have a dedicated SST entry that contains the information about the SST and input values. The VM stores the base address of the sub-trace entry region in the sub-trace entry base address register (`STEBA`). If the number of sub-trace inputs is capped at two, each SST entry requires 256 bits for encoding. SST entries can thus be loaded using a single 256-bit wide load, which has similar latencies as word-sized loads in modern processors [Fog18]. More than two SST inputs can be supported using even wider loads. To find the correct SST entry, the processor uses a hash of the input values to the SST to determine the sub-trace entry address. A larger sub-trace entry region will result in fewer hash collisions. Lower quality hash functions can be used to ensure fast look-ups with a modest increase in hash collisions.

To support efficient skippable sub-traces, we introduce `SETSST` and `SST` instructions, and tagged register write operations that the JIT compiler uses to encode sub-traces. Figure 7.13(a) shows an example assembly snippet of using these features, with the relevant semantics indicated in the comments to Figures 7.13(b–f). The processor supports a number of SST registers (SSTRs), each corresponding to an SST, with fields for input values (`in`), a hash value (`hash`), end PC for the SST (`end_pc`), and return register specifier (`rdest`). At the entry to the JIT-compiled trace, the compiler uses the `SETSST` instruction. As the semantics in Figure 7.13(b) show, this sets the input values in the corresponding SSTR and calculates the hash value of the inputs so far.

```
SETSST 1, r1, r2           #           (b)
...
op1 r1, ...  tagged(1, 0) # nodes      (c)
op2 r2, ...  tagged(1, 1) # j          (c)
...
SST r3, 1, L1        # SST1            (d)
LDM r3, [r1 + r2]    # p1 = nodes[j]   (e)
JEQ r3, 0, Handler1 # guard(p1 != 0)
LDM r4, [r3 + 8]     # p1.class        (e)
# guard(p1.class == UserInstObj)
JNE r4, UserInstObj, Handler2 #        (f)
L1: # If SST1 is skipped, it skips to here.
```

(a)

```
SETSST sst_idx, rin1, rin2, ... rinn:
  SSTR[sst_idx].in[0] = R[rin0]
  SSTR[sst_idx].in[1] = R[rin1]
  ...   # Other inputs
  SSTR[sst_idx].hash = hash(sst_idx,
    ↪ SSTR[sst_idx].in[0], ...)
```

(b)

```
tagged(sst_idx, sst_in):
  # val is being written to reg file.
  SSTR[sst_idx].in[sst_in] = val
  SSTR[sst_idx].hash = hash(sst_idx,
    ↪ SSTR[sst_idx].in[0], ...)
```

(c)

```
SST rdest, sst_idx, offset:
  addr = STEBA + SSTR[sst_idx].hash
  # Wide load from mem or scratchpad.
  retval,idx,in0,in1 = M[addr]
  if idx==sst_idx &
    ↪ in0==SSTR[sst_idx].in[0] &
    ↪ in1==SSTR[sst_idx].in[1] & ...:
    # Valid entry.
    PC = PC + offset
    R[rdest] = retval
  else:
    # Invalid entry.
    SST_IDX = sst_idx
    SSTR[sst_idx].end_pc = PC + offset
    SSTR[sst_idx].rdest = rdest
```

(d)

```
LDM rdest, [addr]:
  protect_addr(SST_IDX, addr)
```

(e)

```
if NextPC ==  SSTR[SST_IDX].end_pc:
  addr = STEBA + SSTR[sst_idx].hash
  retval = R[SSTR[SST_IDX].rdest]
  in0val = SSTR[SST_IDX].in[0]
  in1val = SSTR[SST_IDX].in[1]
  ...  # Other inputs
  M[addr] = retval,SST_IDX,in0val,in1val
  SST_IDX = -1
```

(f)

Figure 7.13: SST ISA Extensions – (a) Usage of SETSST and SST instructions implementing SST1 from Figure 7.10(b). The semantics of different components (b–f) are indicated in the comments. (b) Semantics of the SETSST instruction. Sets up a SST register (SSTR) with the input values at this point. (c) Semantics of tagged register write instructions. It updates the relevant SSTR input value using the value being written to the register file and updates the hash value with the known inputs so far. (d) Semantics of the SST instruction. It calculates the hash value to perform a wide load from either memory or scratchpad. The loaded values contain the return value for the SST entry, the SST index, and the input values. To ensure this entry is valid and there has not been a collision, it checks these values against the values in the SSTR. If the checks have been successful, it skips to L1 if the SSTR valid field is set and writes the return value to the register file. (e) Semantics of marked load instructions, adds the load effective address to the SSIE to be protected. (f) Semantics of sub-trace return. Stores the SST entry to the memory or scratchpad.

Any instruction that produces an input to an SST will be tagged by the compiler. These instructions, in addition to writing to the register file, also update the corresponding SSTRs with the most up-to-date SST input value. By exploiting the fact that the JIT-compiled code is a linear trace, the compiler knows exactly which instructions will produce the input values to the SST. The compiler tags these instructions with the SST index and the SST input number. For example, the operations in lines 3 and 4 in Figure 7.13(a) are both tagged writes for SST1 since they update r1 and r2 respectively, for inputs 0 and 1. This tag can be encoded either inline with the instruction encoding, or separately in a memory region that contains the tag information across multiple instructions. Figure 7.13(c) shows the semantics of tagged instructions. The input value for the corresponding SSTR is then updated with the value that is being written to the register file (line 3). Additionally, the hash is calculated using all of the inputs and the SST index (line 4).

Figure 7.13(d) shows the semantics of the SST instruction. When the SST instruction is executed, the hash field in the SSTR is incremented by STEBA to generate the effective address for the sub-trace entry in the memory or scratchpad (line 2). A wide load is performed using this effective address (line 4). Due to the potential for collisions or invalid entries, the SST index and the input values have to be checked against the values loaded from the sub-trace entry (line 5). If there are no mismatches, then this is a valid entry, so the PC is set to jump over the sub-trace (line 7) and the return value loaded from the sub-trace entry is written to the register file (line 8). Otherwise, the sub-trace will not be skipped. SST_IDX is set to indicate that the processor is in the sub-trace tracing mode (line 11). Additionally, the end PC and the return register specifier are set to finalize a sub-trace entry at the end of sub-trace execution.

If the processor does not skip the sub-trace and it is in the sub-trace tracing mode, the compiler uses marked load operations in the sub-trace (see lines 7 and 9 in Figure 7.13(a)). Figure 7.13(e) shows that these marked loads protect the effective address by sending these to the SST Synchronous Invalidation Engine (SSIE) (described in Section 7.3.4). Finally, upon the end of the sub-trace, the SSTR values and the SST return value are written back to the relevant sub-trace entry in the memory or scratchpad and the execution resumes as normal (see Figure 7.13(f)).

7.3.4 SST Invalidation

Skippable sub-traces are only safe to skip if the values loaded in the sub-trace have not changed since creating the original sub-trace entry. We need special hardware support for monitoring the stores performed by the main processor against the addresses that are read in the sub-trace, and to invalidate the offending sub-trace entries in case of a match. We use a hybrid of synchronous and asynchronous invalidation engines. The invalidation of sub-traces is only timing critical if the sub-trace is going to be executed soon after the store that requires invalidation. However, a typical execution may contain many JIT-compiled traces, each of which have sub-traces that need to be checked for invalidations. These sub-traces will not be executed in the near future, so these entries can be checked asynchronously. We use the helper processor from Figure 7.6 to perform these invalidation checks. The store addresses performed by the main processor are forwarded to the helper processor using the memory request/response snooping network. The helper processor, just like in profiling, uses software to check against any matches with addresses read in sub-traces. The software can be conservative: it is always a valid execution to conservatively invalidate an entry. Not skipping the sub-trace body will only cause a minimal performance degradation compared to executing the operations in the original (before recompilation) trace.

While checking for invalidations in the sub-traces in other traces can be handled asynchronously, it is much more timing critical to check the sub-traces that belong to the currently executing trace. Because of this, we propose using a special hardware unit called the SST Synchronous Invalidation Engine (SSIE) to check for matches. SSIE is attached to the memory request bus, just like the memory request/response snooping network. It consists of a number of equality checkers and Bloom filters [Blo70]. The equality checkers are used to check the store address against either an exact load address from the sub-trace or a range of addresses. Equality checkers can also be used as pre-filters to the Bloom filters, reducing the false positive rate. In the sub-trace tracing mode, the marked load addresses will be inserted into the available equality checkers and Bloom filters on a first-come first-served basis. The general approach in assigning these units is to first check if the new address already matches one of the equality checkers or Bloom filters, and if so, use those units. If not, and if there are equality checkers that have not yet been assigned to any entry, then add the address to that equality checker. If there are no equality checkers available, pick a Bloom filter using a round-robin basis and add this address into that Bloom filter. SSIE also contains an entry table (ET) indexed by the equality checker and Bloom filter ID, which returns the corresponding sub-trace entry address (STEA). In case any of these units find a match when checking a store, they will do a lookup in the ET, and find the corresponding STEA. To invalidate these sub-trace entries, both the helper processor and SSIE write zero to the PC field of the sub-trace entry using the memory request bypass channel.

7.4 Evaluation

In this section, we describe the methodology for evaluating sub-trace skipping. We then show the results in terms of dynamic instruction reduction in JIT-compiled regions, whole program analysis, and various sensitivity studies. We finally describe a prototype chip taped out in a TSMC 28nm process.

7.4.1 Methodology

We evaluated sub-trace skipping in the context of the RPython framework described in Chapter 2. In RPython, we implemented a mechanism to encode the object dereferences using an in-memory representation. We allocate the per-trace profiling region in the virtual memory system, as described in Section 7.3.1. We also implemented recompilation as an additional optimization pass at the end of the optimization pipeline. We implemented a number of different sub-trace growing algorithms as described in 7.3.2. We also made changes to the instruction generation and register allocation parts of the compiler backend. The RPython register allocator only works with linear traces, so it does not support basic block merge points. From the perspective of code generation, sub-traces look like a conditional jump, so sub-traces require a register allocator that can handle basic block merge points. We modified the register allocator to allow reasonable sub-trace register allocation with as few spills as possible. In the instruction generator, we encoded the `SST` instruction using an x86 `NOP` instruction. Because the usual x86 addressing modes can be used with the `NOP` instruction, and the address values for a `NOP` are normally ignored by the processor, we used the additional bits in the address to encode the register operands, the destination register, and the offset.

An important design goal of meta-tracing JITs is to provide a common JIT infrastructure across multiple frontend languages. By targeting meta-tracing JIT VMs, sub-trace skipping can accelerate multiple programming languages. To illustrate this, we evaluate SST on VMs for two different dynamic languages: Python and Racket. The VMs for these languages, PyPy [RP06] and Pycket [BBH+15] respectively, are written using the RPython language and make use of RPython meta-tracing JIT compiler. As Chapter 6 shows, PyPy is the fastest implementation of Python and Pycket is on par in terms of performance with the reference JIT-optimized VM for Racket.

We use Pin [LCM+05] to model the SST hardware and helper processor. Our modified JIT runtime communicates the address of the profile region to the functional model using a `NOP`

Core	1 x86 OOO core at 2.4GHz
L1I cache	32KB, 4-way, LRU, 3-cycle latency
L1D cache	32KB, 8-way, LRU, 4-cycle latency
L2 cache	256KB, 8-way, LRU, 10-cycle latency
L3 cache	8MB, 16-way, LRU, 4 banks, 46-cycle latency
DRAM	DDR3-1333-CL10, 40-cycle controller latency

Table 7.2: ZSim Microarchitectural Parameters

instruction. The modeled helper processor software tracks object dereference instructions that originate from the JIT-compiled code for value patterns. Once a per-trace trip count has been reached, the model populates the profile region with ODL information and triggers a recompilation. We also model the semantics of the `SST` instructions in Pin, as explained in Section 7.3.3. We model three different configurations for SSIE: ideal (SST-I), 1-Bloom-Filter SSIE (SST-b1), and 8-Bloom-Filter SSIE (SST-b8). With our Pin-based methodology, we can retrieve statistics for dynamic instructions in various parts of the VM execution (similar to the methodology presented in Chapter 6) and SST hardware performance counters. As the SST technique targets the JIT-compiled code, we focus on dynamic instruction count reduction while executing JIT-compiled code.

In addition, we use ZSim to model the microarchitectural performance of SST [SK13]. In addition to the functional changes described above, we also had to modify ZSim to model the `SST` instructions. A Pin basic block is a single-entry-single-exit code unit. Thus, if a basic block runs, then all instructions that belong to it are executed. ZSim leverages this by simulating at basic block granularity. It records loads and stores within a basic block of native execution, and then ZSim uses this information to drive the timing models. However, since we overloaded `NOP` instructions to encode the `SST` instruction, to model skipping over the sub-trace, we re-steer the execution path using Pin in the middle of a basic block. Thus, only part of this basic block instrumented by ZSim runs, as Pin discards the original basic block and creates a new one in such cases. In order to cope with this, we break basic blocks into sub-blocks after each `SST` instruction. We also treat `SST` instruction as a conditional branch, so we can re-use the existing misprediction penalty calculating mechanism in ZSim to model it as a control instruction. The microarchitectural parameters we have used for ZSim are shown in Figure 7.2.

We use a subset of benchmarks from two suites: the PyPy Benchmark Suite [pypb] and the Computer Languages Benchmarks Game (CLBG) [Gou] to evaluate our technique. Both of these benchmark suites have benchmarks for the Python language, and CLBG also contains Racket

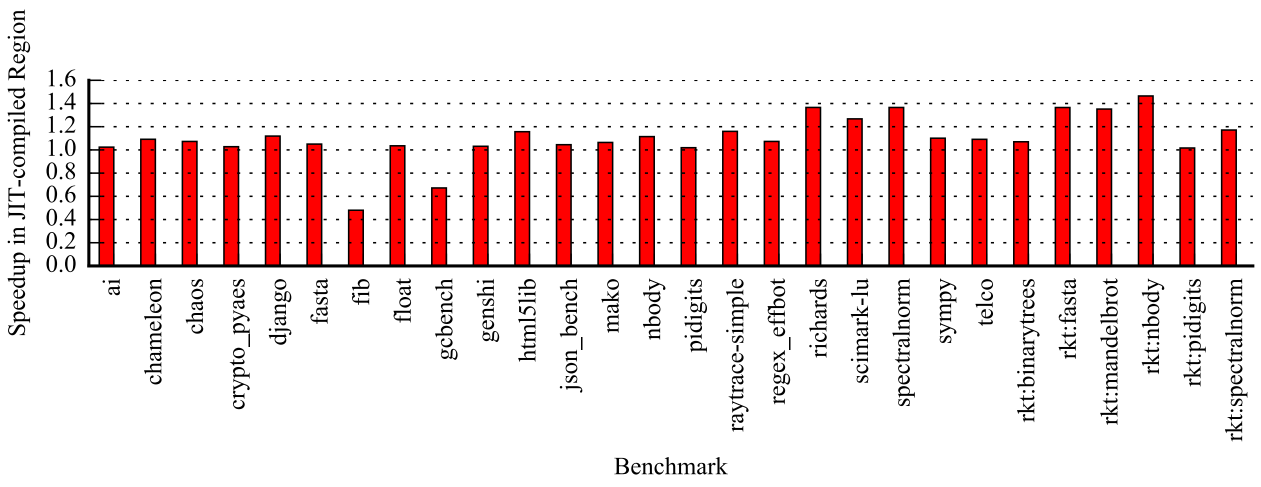


Figure 7.14: Speedup in JIT-Compiled regions for SST-I in Dynamic Instructions – The benchmarks that start with `rkt:` are the Racket programs, remaining are Python programs.

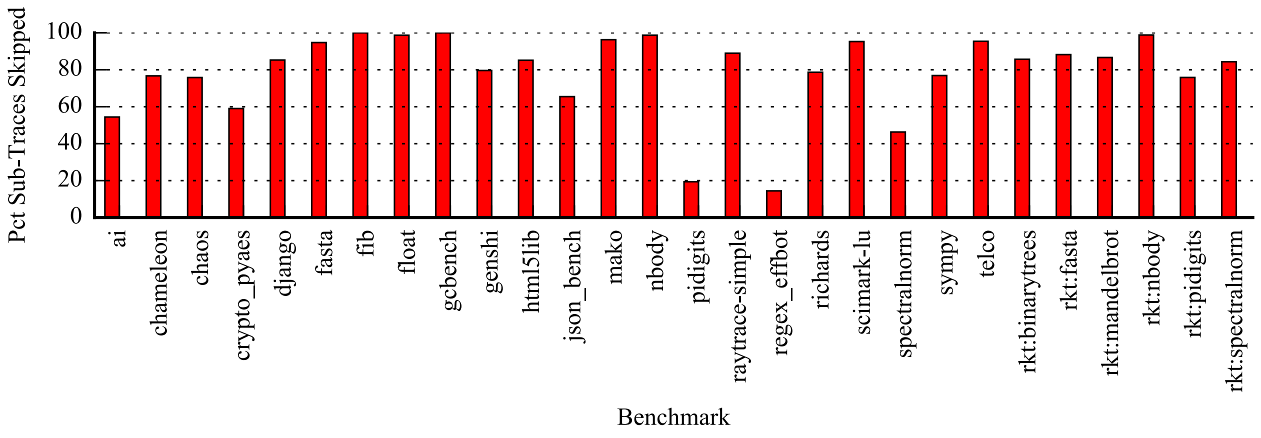


Figure 7.15: Percentage of Skipped Sub-Traces – The benchmarks that start with `rkt:` are the Racket programs, remaining are Python programs.

benchmarks. Benchmarks such as `django` (a web template framework), `html5lib`, `chameleon`, and `json_bench` are web-flavored non-numerical real-world benchmarks that are generally representative of popular Python workloads.

7.4.2 Reduction in Dynamic Instructions in JIT-Compiled Regions

The goal of the sub-trace skipping technique is to reduce the dynamic instruction count in JIT-compiled traces. This section presents the results of these reductions. However, Chapter 6 shows that non-JIT-compiled regions can dominate the execution time for some of the benchmarks.

Benchmark	NR JDI	SST-I JDI	Speedup	# STs	# LU	# Skip	# Inv	# STEs
fasta	131M	96.5M	1.36	133	7.26M	6.40M	813K	850K
mandelbrot	7.13B	5.27B	1.35	67	58.1M	50.4M	7.28M	7.76M
nbody	156M	107M	1.46	150	8.56M	8.46M	526	50.2K
pidigits	8.05M	7.92M	1.02	85	49.4K	37.5K	10.7K	11.9K
spectralnorm	1.65B	1.41B	1.17	54	393K	331K	49.3K	61.5K

Table 7.3: Racket VM performance – Dynamic instructions in the JIT-compiled region (JDI) and speedup in the JIT-compiled region for the no-recompile (NR) and ideal sub-trace skipping (SST-I) configurations. SST statistics for the SST-I configuration: # STs = number of sub-traces, # LU = number of sub-trace lookups, # Skip = number of sub-trace skipped, # Inv = number of sub-trace entries invalidated, # STEs = number of sub-trace entries.

Furthermore, recompilation induces additional execution overhead, which are not captured in results in this section. We evaluate those overheads in Section 7.4.3.

Figure 7.14 shows the speedup of the SST-I configuration in the JIT-compiled region compared to the no-recompile baseline. The benchmarks that have `rkt:` in front of their names are the Racket-language benchmarks running on Pycket. The remaining are Python programs running on PyPy. With the exception of two benchmarks, `fib` and `gcbench`, all other applications have reductions in dynamic instructions. While the speedup due to dynamic instruction reduction in many of the Python benchmarks is only a few percent, there are also many benchmarks, including `chameleon`, `django`, `html5lib`, and `telco`, that have speedups of around 10%. A few benchmarks show even higher speedup, such as `richards` and `spectralnorm`, with more than 35% improvement. Also worth noting is the Racket benchmarks such as `rkt:fasta`, `rkt:mandelbrot`, and `rkt:nbody` see even larger improvements. Table 7.3 shows additional statistics for these Racket benchmarks. It is possible that the reason Racket benchmarks have more benefit is because Pycket is not as well optimized, so our hardware might have a more significant impact on such interpreters.

`fib` and `gcbench` perform especially poorly on SST, because these benchmarks are control-flow intensive. These applications do not have clearly defined loops, so the tracing-JIT traces different control flow paths arbitrarily. Because of this, we see that pathological paths can be traced when enabling recompilation. Future work may look into improving the behavior of these pathological control-flow intensive benchmarks.

We also present how often SSTs are skipped over (see Figure 7.15). The data shows a very wide range of skip rates. The sub-trace skipping percentages do not seem to directly correlate with the speedup in the JIT-compiled regions. However, it shows that there might be room for

	Baseline				NR				RO				SST-I				SST-b8				SST-b1			
Benchmark	**JDI**	**JC**	**DI**	**C**	**JDI**	**JC**	**DI**	**C**	**JDI**	**JC**	**DI**	**C**	**JDI**	**JC**	**DI**	**C**	**JDI**	**JC**	**DI**	**C**	**JDI**	**JC**	**DI**	**C**
ai	9.2	9.9	15.7	16.3	9.3	10.0	15.9	16.4	9.9	12.1	16.7	19.7	9.0	11.7	15.8	19.4	9.2	11.8	15.9	19.5	9.2	11.9	16.0	19.5
django	5.4	6.7	16.2	16.9	5.6	6.8	16.7	17.3	5.9	7.9	17.4	20.3	5.0	7.5	16.5	19.9	5.0	7.4	16.5	19.8	5.6	7.7	17.1	20.1
fasta	11.2	11.7	16.0	16.1	11.6	11.7	16.4	16.1	10.1	12.9	15.0	17.6	9.1	12.4	14.0	17.4	9.1	12.4	14.0	17.4	9.2	12.5	14.1	17.3
nbody	7.8	7.4	15.1	14.3	7.9	7.3	15.2	14.3	7.9	7.4	15.3	15.1	7.1	7.4	14.5	14.7	7.1	7.4	14.5	14.7	7.2	7.7	14.6	14.9
richards	14.3	11.6	15.8	13.2	14.9	11.9	16.6	13.7	14.8	12.8	17.6	15.5	11.0	11.7	13.7	14.4	11.6	11.7	14.3	14.4	12.4	12.3	15.2	15.0
spectralnorm	13.3	10.7	15.4	13.6	14.2	10.8	16.4	13.7	14.4	12.0	16.8	15.2					13.5	11.7	15.8	14.9	13.5	11.7	15.8	14.9
telco	6.4	10.0	15.0	19.0	8.5	13.4	17.6	23.3	8.9	16.2	18.8	27.8	8.0	15.2	17.8	27.0	8.0	15.2	17.9	27.0	8.3	15.6	18.2	27.4

Table 7.4: Python VM Performance – Dynamic instructions (JDI) and cycles (JC) in the JIT-compiled region, dynamic instructions (DI) and cycles (C) in the entire execution for baseline Pypy (Baseline), no-recompile (NR), recompile only but no-skipping (RO), ideal sub-trace skipping (SST-I), realistic sub-trace skipping with 8 equality checkers and 1 512-bit bloom filter (SST-b1), and realistic sub-trace skipping with 8 512-bit bloom filters (SST-b8). All dynamic instructions and cycles countings are in billions. The SST-I configuration for spectralnorm did not finish due to a simulation issue.

improvement in the profiling algorithm for more accurate ODL detection. In some benchmarks, such as `pidigits`, a high percentage of the object dereferences that the profiler identifies to have locality turn out not to do so later in the algorithm. Future work may look at improving the profiling to improve the percentage of skipped sub-traces.

7.4.3 Whole-Program Analysis

While the SST technique improves the dynamic instruction count in the JIT-compiled regions, the VM also has to perform additional work to recompile traces. To characterize these overheads and evaluate how long a benchmark has to run in order to amortize the cost of recompilation and benefit from SST, we ran a few of the benchmarks for more than 15 billion dynamic instructions (a few seconds of simulated time). Table 7.4 shows the results of these runs. We also evaluated these runs on the ideal SSIE (SST-I) and more realistic 1- and 8-Bloom-filter SSIE configurations. We compare these simulations to three different baseline configurations. The first baseline configuration is PyPy with our changes disabled. The no-recompile (NR) configuration adds a recompilation guard to the traces. However, we configure Pin not to recompile in this setting. Recompile-only (RO) configuration only performs profiling and recompilation, but actually does not skip the sub-traces.

These results show that the dynamic instructions in the JIT-compiled region (the JDI columns) is better compared to the baseline configuration using an ideal SSIE, with the exception of one benchmark. We can also see that the NR and RO configurations have non-trivial overheads

Benchmark	# STs	# LU	#Skip	#Inv	#STEs
ai-SST-I	250	502M	272M	227M	227M
ai-SST-b1	250	502M	220M	179M	179M
ai-SST-b8	250	502M	228M	152M	152M
django-SST-I	298	272M	232M	34M	35M
django-SST-b1	298	272M	94M	515K	515K
django-SST-b8	298	272M	229M	2M	2M
fasta-SST-I	56	279M	277M	2M	2M
fasta-SST-b1	56	279M	252M	5M	5M
fasta-SST-b8	56	279M	275M	2M	2M
nbody-SST-I	44	284M	281M	4M	4M
nbody-SST-b1	44	284M	281M	18K	18K
nbody-SST-b8	44	284M	281M	199	314
richards-SST-I	598	1237M	973M	124M	124M
richards-SST-b1	598	1237M	616M	6M	6M
richards-SST-b8	598	1237M	864M	16M	16M
spectralnorm-SST-I					
spectralnorm-SST-b1	43	546M	298M	39K	39K
spectralnorm-SST-b8	43	546M	298M	41K	41K

Table 7.5: Python VM SST Statistics – # STs = number of sub-traces, # LU = number of sub-trace lookups, # Skip = number of sub-trace skipped, # Inv = number of sub-trace entries invalidated, # STEs = number of sub-trace entries. The SST-I configuration for spectralnorm did not finish due to a simulation issue.

compared to the baseline in the JIT-compiled region. These overheads are due to the additional recompile guards (for NR) and slightly worse instruction generation and register allocation (for RO). The dynamic instruction counts for the overall execution (the DI columns) show that half of the benchmarks perform better with an ideal SSIE compared to the baseline. This means that for these benchmarks, the break-even point to amortize the additional recompilation overhead is reached within a few seconds. For workloads that can be expected to run a long time (e.g., server workloads), this is promising. However, we observe a slowdown when we look at the total cycle count (the C columns). Even more surprising is that the NR baseline also has up to 20% slowdown in terms of cycles compared the baseline with no recompile checks even though the dynamic instruction count is similar. This might be an artifact of how we check and perform recompilations, and future work may explore techniques to improve this overhead.

Comparing different SSIE configurations, we see that the performance of SST-b1 is actually similar to both the ideal configuration and SST-b8. This indicates that the additional hardware requirements can be kept at a minimum and achieve similar performance as an ideal SSIE. Table 7.5

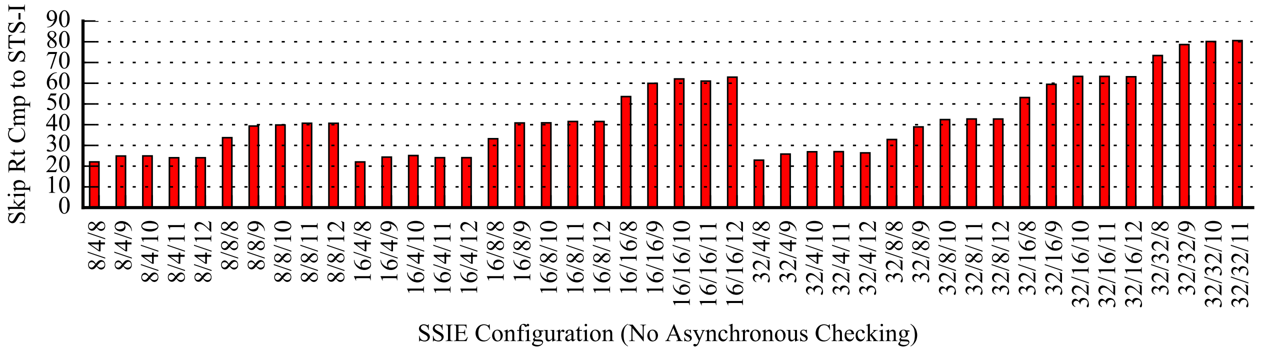


Figure 7.16: Skip Rate Percentage Compared to SST-I (No Asynchronous Checking) – For a selection of SSIE configurations, running richards (number of equality checkers/number of Bloom filters/Bloom filter size in 2^N).

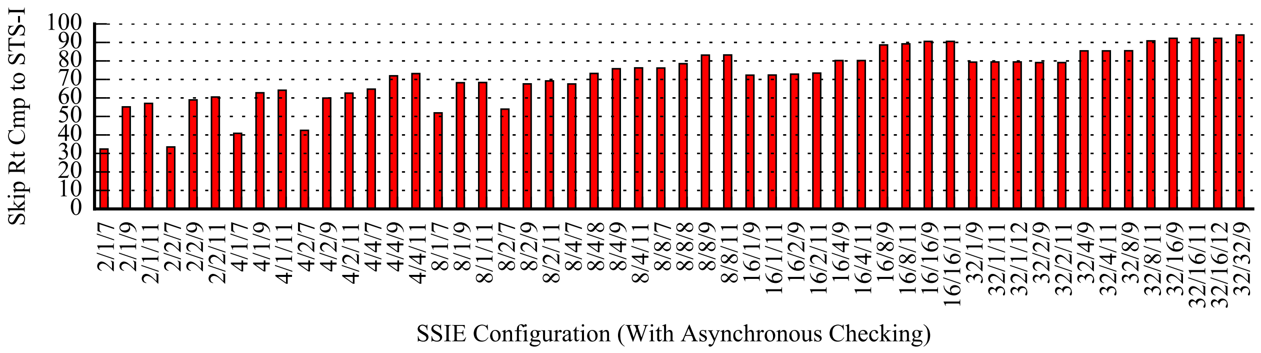


Figure 7.17: Skip Rate Percentage Compared to SST-I (With Asynchronous Checking) – For a selection of SSIE configurations, running richards (number of equality checkers/number of Bloom filters/Bloom filter size in 2^N).

shows the SST statistics of these benchmarks with different SSIE configurations. Using SST-b1 configuration can reduce both the number of sub-trace entries and the number of invalidations while achieving similar overall performance benefits in the JIT-compiled region.

7.4.4 SSIE Sweep Case Study

We explore the impact of the SSIE microarchitecture on the sub-trace skipping rates. Figure 7.16 shows a sweep of the SSIE architectures without an additional helper processor to do different-trace checking. We see that the skip rates are quite low in this configuration unless the number of Bloom filters and equality checkers are increased to substantial amounts. To get 60% hit rate, it is necessary to use 16 Bloom filters, and 16 additional equality checkers. However, if we use a helper processor to check for non-critical addresses, it is possible to drastically reduce the pressure on the SSIE (see Figure 7.17). In this configuration, it is possible to get to 70% of the skip rate of an ideal SSIE with a single 512-bit Bloom filter and 8 additional checkers. So, the asynchronous invalidation checking

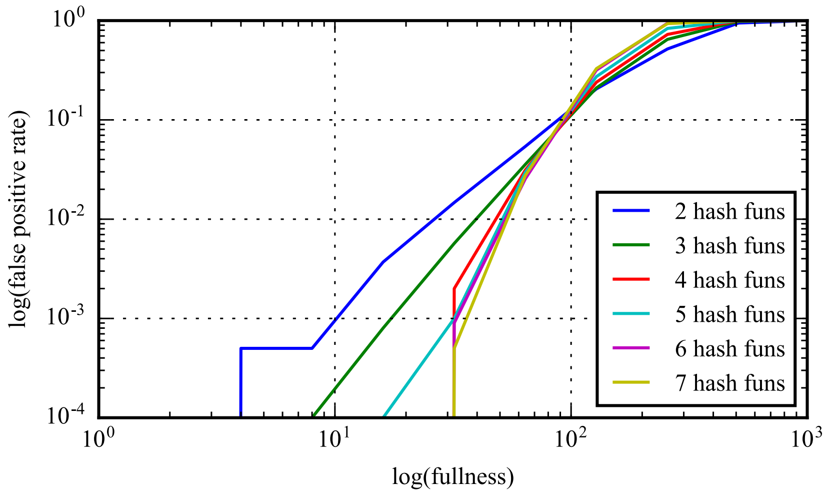


Figure 7.18: 512-bit Bloom Filter False Positive Rates – Shows a sweep of how many items the Bloom filter contains and the false positive rate for different number of hash functions.

using a helper processor is necessary to keep the additional area requirements low while skip rates high. The SST-b1 and SST-b8 designs use 512-bit Bloom filters.

Bloom filters operate by using a number of hash functions and setting or checking the bits indexed by these hashes. The number of hash functions and how many items have already been inserted to the Bloom filter affect the false positive rate of the Bloom filter. Figure 7.18 shows a sweep of the number of items inserted to a 512-bit Bloom filter and the false positive rates for a sweep for the number of hash functions. For example, a 2-hash-function Bloom filter that has already 64 items inserted has 5.4% of the searches for items that are not in the Bloom filter are falsely identified as contained in the filter, whereas the false positive rate drops to 3.5% and 3.1% for 3 and 4 hash functions respectively. We find that a 4-hash-function 512-bit Bloom filter is a good balance of area overhead and false positive rates.

7.4.5 Recompile Options Case Study

As Section 7.3.2 has shown, there is a rich design space in growing sub-traces. Table 7.6 shows the statistics of different recompile options. We evaluate three options, s = whether additional pure operations can be added to the sub-trace that does not use the output of the dereference but instead uses the input value, n = whether additional pure operations can be added that requires the addition of a *new* input, and c = whether multiple dereferences can be included in the sub-trace in a chained

Benchmark	RO	SST-I JDI	# STs	# LU	# Skip	# Inv
ai		497M	140	17.6M	12.3M	5.29M
ai	s	499M	144	19.9M	12.3M	7.52M
ai	sn	497M	181	27.6M	15.0M	12.3M
ai	c	510M	232	27.7M	13.3M	13.4M
ai	cs	509M	224	27.8M	10.7M	15.8M
ai	csn	511M	227	28.0M	6.68M	19.9M
django		144M	227	6.48M	6.39M	13.7K
django	s	144M	229	6.61M	6.53M	13.7K
django	sn	141M	188	5.89M	4.46M	1.27M
django	c	145M	253	7.89M	6.73M	967K
django	cs	141M	191	5.49M	5.41M	14K
django	csn	139M	177	5.16M	5.07M	17K
fasta		200M	47	4.88M	4.86M	27.7K
fasta	s	200M	47	4.88M	4.86M	27.7K
fasta	sn	194M	58	7.37M	6.98M	157K
fasta	c	198M	61	7.82M	7.79M	27.9K
fasta	cs	197M	53	6.79M	6.05M	723K
fasta	csn	189M	49	8.42M	7.33M	852K
rkt:fasta		94M	122	6.38M	6.31M	68.2K
rkt:fasta	s	94M	122	6.38M	6.31M	68.2K
rkt:fasta	sn	96M	133	7.26M	6.40M	813K
rkt:fasta	c	92M	138	6.32M	6.22M	101K
rkt:fasta	cs	90M	151	6.04M	5.94M	97.4K
rkt:fasta	csn	95M	158	6.65M	4.93M	1.63
richards		193M	520	20.2M	15.7M	2.28M
richards	s	193M	520	20.2M	15.7M	2.28M
richards	sn	193M	568	21.1M	16.6M	2.25M
richards	c	197M	505	17.5M	12.9M	2.16M
richards	cs	206M	431	14.1M	8.61M	2.98M
richards	csn	206M	420	14.0M	8.50M	2.95M
telco		1.09B	516	32.2M	31.2M	1.04M
telco	s	1.09B	516	32.2M	31.2M	1.04M
telco	sn	1.08B	577	35.7M	34.0M	1.60M
telco	c	1.08B	499	30.1M	29.1M	987K
telco	cs	1.07B	392	27.5M	26.2M	1.34M
telco	csn	1.07B	458	31.0M	26.5M	4.49M

Table 7.6: Impact of Recompilation Options – RO = recompile options (s: share input, n: new input, c: chained loads); JDI = number of dynamic instructions in JIT; # LU = number of sub-trace lookups; # Skip = number of sub-trace skips; # Inv = number of sub-trace entries invalidated.

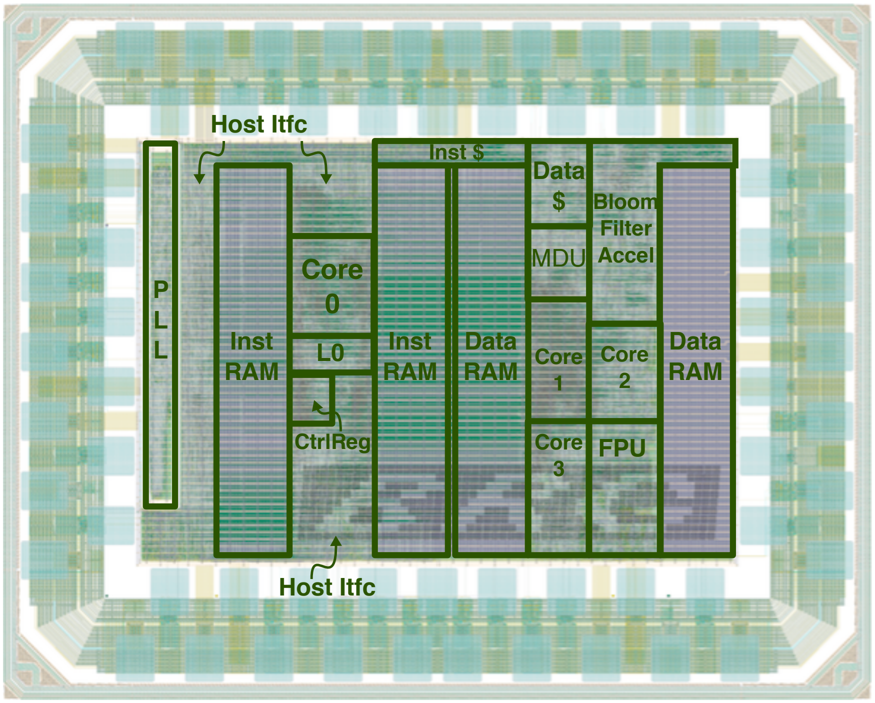


Figure 7.19: Layout of BRGTC2 – The Batten Research Group Test Chip 2 (BRGTC2) was produced using TSMC 28nm process. The chip contains a prototype Bloom filter accelerator similar to the proposed SSIE.

fashion. Interestingly, we see that there is no single setting that performs the best. For the most benchmarks, sn performs consistently well, but in a few benchmarks, csn and cs outperform sn. The important insight is that as sub-traces grow, more instructions can be skipped, which is good for performance. However, if any of the loads in the SST are invalidated, the entire sub-trace entry would need to be invalidated as well. So we observe that the more aggressive sub-trace growing algorithms tend to suffer more from invalidations due to this effect.

7.4.6 Prototype Chip Tapeout and Area Analysis

To evaluate the timing and area overhead of SSIE, we have implemented a parametrizable Bloom filter design in RTL. This Bloom filter uses a parametrizable number of bits for storage and hash functions implemented using constant-value multiplication. The constant-value multiplication technique performs a multiplication with a constant prime number, which is converted to efficient constant-value shifts and adders during logic synthesis. This design is similar to the one proposed by Lyons and Brooks [LB09], and we did not observe a significant difference in false positive rate compared to using cryptographic hash functions.

Component	**Area (µm^2)**	**Percent**
Entire design	388,700	100.0
Instruction cache (32KB)	131,916	33.9
Data cache (32KB)	130,774	33.6
Processors (4×)	44,097	11.3
Host interface	35,238	9.1
Bloom filter accelerator (4× 256-bit, 3-hash)	19,767	5.1
Shared FPU (1×)	11,291	2.9
L0 instruction cache	6,618	1.7
Shared Multiply/Divide Unit (1×)	4,874	1.3
Control Registers	1,930	0.5
Network	1,280	0.3

Table 7.7: BRGTC2 Area Breakdown

# Bits	**# Hash functions**	**Area (µm^2)**
256	1	3081
256	2	4042
256	3	5005
256	4	6105
512	2	5319
512	3	6601
512	4	7835
1024	2	7794
1024	3	9792
1024	4	11908
1024	5	14038

Table 7.8: Bloom Filter Area Sweep – Shows various Bloom filter configurations synthesized using TSMC 28nm process and 1ns clock period.

In Spring 2018, our group built the Batten Research Group Test Chip 2 (BRGTC2) that incorporated a number of techniques from ongoing projects, including a prototype Bloom filter accelerator [TJAH$^+$18]. We designed this chip using PyMTL [LZB14, JIB18] in a TSMC 28nm process, with a maximum clock frequency of 500MHz. The chip includes four in-order scalar processor cores that make use of the sharing architecture [Sri18], where the caches, multiply/divide unit, and the FPU are shared among the cores. We attached four 256-bit, 3-hash Bloom filter accelerators to snoop the memory request buses for each core. The processors can use an accelerator interface and specialized instructions to insert entries, search, reset, and configure the Bloom filter to snoop memory reads, writes, or both. Figure 7.19 shows the annotated layout of the taped-out chip, and Table 7.7 shows the area breakdown.

The area of the Bloom filter accelerator shows that four small Bloom filters only occupy 5% of a very small chip area, where each accelerator is roughly equal in size to a 32-bit iterative integer multiply/divide unit. We have further synthesized other Bloom filter accelerator designs using the same process, for 1GHz clock target, as seen in Table 7.8. Both the number of Bloom filter bits and the number of hash functions affect the area, and the area overhead of an additional hash function roughly doubles going from a 256-bit to a 1024-bit Bloom filter. The 512-bit 4-hash Bloom filter that SST proposes is less than 8000 μm^2, which is a very modest area overhead of around 70% of an FPU.

7.5 Related Work

Polymorphic inline caches [HCU91] can be seen as the most direct pure-software equivalent of subtrace skipping. It is used only to speed up the lookup of methods at the method call site, by keeping a small cache directly in the generated machine code. The cache contains of pairs of precise receiver types and called methods. Every time a call to an unknown receiver type is seen at a call site, a slow method lookup is performed, and a new pair of that receiver type and the found method is added to the cache. When a method is re-defined, the pairs need to be removed from all affected method call sites, which is all done purely in software with a write barrier specific to method redefinitions. This technique is useful, but only used specifically for method lookups, not for other sources of value locality, due to the cost and complexity of the related write barrier.

Relatedly, a lot of powerful JIT optimization approaches optimize the generated machine code based on an interpreter-level assumption and will have to invalidate the generated code once the assumption is broken later. A typical example is Java Virtual Machines assuming that class loading is rare, and thus optimizing based on a whole world assumption [SBC00]. If later a subclass of an existing class is loaded, generated machine code that assumed that some method did not have any overrides will have to be deoptimized [HCU92]. This approach only works when the write barrier that is needed to detect situations where deoptimizations might be necessary only need to be applied to a tiny percentage of objects (for example classes) and not to arbitrary memory locations. Furthermore, unlike the SST technique, the interpreter has to hard-code these assumption heuristics, which can be cumbersome. On the other hand, an advantage of pure software approaches is that deoptimization conditions can be arbitrarily complex, for example to check whether the content of a complicated hashmap changed.

There have been a number of proposals to use software/hardware co-design to improve the performance of dynamic languages. Three prior work are the closest proposals sub-trace skipping: Checked Loads proposes ISA extensions for JIT-compiled code to perform loads and type checks at the same time [AFCE11]; ShortCut proposes an ISA extension to more efficiently perform type dispatch [CSGT17]; and Dot et al. propose a scheme to remove monomorphic class checks altogether using profiling [DMG17]. All three of these techniques specifically target certain scenarios of class checks, however sub-trace skipping is more general and can identify class check opportunities in addition to other object dereference localities. Furthermore, Dot et al. use software profiling (which can introduce overheads) in mid-tier JIT compilation (not final-tier) to discover monomorphic types, whereas sub-trace skipping uses asynchronous no-overhead profiling that can work at final-tier JIT-compiled code. Checked Loads and ShortCut do not use profiling to identify the patterns to optimize.

Other prior work on software/hardware co-design includes changes to the memory system for more efficient dynamic memory management [TGS18], hardware support for faster deoptimizations [SCGT19], hardware acceleration of garbage collection [CTW05, MAK18, TIW11, Mey06, HTCU10, JMP09], and techniques to accelerate the interpreter only [MO98, Por05, KKK$^+$17, RSS15], and custom hardware accelerators for parts of the VM functionality [GSL17]. Many of these techniques are orthogonal to sub-trace skipping.

Value prediction [GM98] and instruction reuse [SS97, YG00] have been proposed to exploit data locality. Value prediction predicts the output of an instruction by tracking its history outputs, and speculatively pass this predicted value to later instructions. This instruction will still be executed and the predicted value is checked against the actual output. If there is a midprediction, the processor needs to squash and restart. Unlike value prediction, instruction reuse is a non-speculative method. It stores inputs and outputs of an instruction into hardware tables. During fetching, the actual inputs are compared with known ones. If there is a match, stored outputs will be written to destinations and this instruction is ready to commit. Later proposals [HL99, GTM99] extended the idea of instruction reuse to basic block and trace granularity. Unlike previous works, which rely on the hardware to discover and exploit data locality, our technique leverages a software/hardware hybrid method with asynchronous profiling and software recompiling. We only use hardware for time critial tasks such as SST look ups and store-protection.

7.6 Conclusion

We have presented skippable sub-traces, which is a software/hardware co-design technique to exploit load-value locality in object dereferences, in tracing-JIT VMs for dynamic languages. We present a characterization of this locality in the context of RPython meta-tracing JIT VM framework. We propose the SST technique consisting of hardware profiling, recompilation, execution, and invalidation. We evaluate the design and present that the SST design is promising in speeding up the JIT-compiled region of multiple dynamic languages by up to 40% in terms of dynamic instructions. However, we also highlight that this technique can have non-trivial overheads and the performance gains might be offset by these overheads. We have also successfully taped out a chip in TSMC 28nm process to prototype some of the techniques presented in this chapter.

CHAPTER 8
CONCLUSION

This thesis presented co-optimizing hardware design and meta-tracing JIT VMs from two angles: accelerating hardware simulations using meta-tracing JIT VMs and building hardware accelerators for meta-tracing JIT VMs. As Moore's Law slows down, it creates an urgent need for agile hardware design methodologies and acceleration techniques for dynamic languages. Both of these challenges can be addressed using approaches similar to those presented in this thesis.

8.1 Thesis Summary and Contributions

This thesis began by motivating the need for new agile hardware design methodologies. As Moore's Law and single-threaded performance improvements slow down, in order to address computational challenges of the future, a more wide-spread use of domain specific hardware is needed. Relatedly, there is a need for new execution mechanisms beyond software-only techniques for dynamic-language JIT VMs. These two needs form the basis of the two parts of this thesis. Then I provided an extensive background on dynamic languages, just-in-time compilation, classical JIT optimization techniques, and two new approaches that allow more productive development of JIT VMs for new dynamic languages: partial evaluation and meta-tracing. Meta-tracing JIT VMs form an important theme for this thesis. Meta-tracing JIT VMs and agile hardware design can be used to address the two needs mentioned earlier. While focusing on partial evaluation instead of meta-tracing could have also been an interesting direction, some of the reasons to choose meta-tracing include the easier use compared to partial evaluation, and the fact that PyPy, currently the fastest VM for Python uses the RPython meta-tracing JIT framework.

Part I of this thesis proposed high-performance agile hardware simulation methodologies using meta-tracing JIT VMs. I explored opportunities and challenges in each of functional level, cycle level, and register-transfer level modeling. In each of these challenges, the meta-tracing JIT compilation technology can be repurposed to yield very high performance simulators, often as fast as, and sometimes faster than, purpose-built frameworks that take many programmer-decades to develop. Furthermore, the techniques proposed in this part also increase developer productivity and agility, by enabling greater flexibility in modifying ISAs, easier design space exploration, taking advantage of highly parameterized designs, and better testing.

Part II of this thesis explored opportunities to use a software/hardware co-design approach to improve the performance of meta-tracing JIT VMs. I initially performed an extensive cross-layer workload characterization of the RPython framework for two different interpreters: Python and Racket. While I did not identify a silver bullet that will magically improve the performance by orders of magnitude, there are many smaller sub-problems that can be improved for parts of the execution. One promising avenue is that object dereferences in JIT-compiled traces can often have value locality: the same load address and loaded data are encountered while executing the load throughout the execution. Using this insight, I proposed a software/hardware co-design technique of identifying object dereference locality, recompiling these traces to allow sub-trace skipping, and skipping these sub-traces with object dereferences if hardware can guarantee that it is safe to do so.

To reiterate the major contributions of this thesis:

- I demonstrated meta-tracing JIT VMs can be used to speed up agile FL simulation of processors.
- I extended agile FL simulation using JIT fast-forward embedding and instrumentation for sampling-based agile CL simulation of computer systems.
- I presented JIT-aware hardware generation and simulation framework (HGSF) and HGSF-aware JIT mechanisms for accelerating agile RTL hardware simulations using meta-tracing JIT VMs.
- I presented a novel methodology to study meta-tracing JIT VMs and allow experimenting with hardware acceleration techniques.
- I performed an extensive cross-layer workload characterization of meta-tracing JIT VMs.
- I identified and quantized object dereference locality (ODL) and intermediate-representation-level value locality (IVL) in JIT-compiled code in the context of meta-tracing JIT VMs.
- I proposed a software/hardware co-design approach to exploit ODL and IVL.

8.2 Future Work

Application-level architectural descriptions in Pydgin – Pydgin allows a pseudo-code-like interface by using the RPython language as the embedded DSL to describe ISAs. This approach enabled us to take advantage of high-performance JIT compilation while retaining simple architecture descriptions. However, one drawback of this approach are the translation times. RPython needs to perform whole-program analysis of the VM to determine all of the types, which can take more than an hour for the Python interpreter and fifteen minutes for Pydgin. The length of translation times can seriously hinder quick iteration when there are changes in the ISA specification. As a future work, mechanisms shorten this iteration loop could be explored. One possible direction is to use the high-performance mixed-language JIT capability of the RPython framework. Pydgin can be implemented as a Python module, and this combined interpreter can then be translated. Any guest instruction that is supported by Pydgin directly will be able to achieve Pydgin levels of performance. New instructions can be described at the application level in Python, and the JIT-optimized Python interpreter can execute those semantics. Pydgin can further expose a high-performance API to the application level, similar to HGSF-aware JIT approach we used for the Mamba project. The instructions that are described at the application level will likely be slower than Pydgin-level descriptions. However, once they are mature, these new instruction semantics can eventually be migrated into RPython, taking full advantage of the meta-tracing JIT compiler.

Host pages in Pydgin – A source of inefficiency in Pydgin is the memory representation. The work presented in Chapter 3 does not model a memory management unit (MMU). The memory is represented as a large RPython list, and there is no address translation necessary. While this approach is fast, the memory usage by the simulator can be significant, regardless of the memory needs of the application. The version of Pydgin that has self-modifying code support models an MMU just for invalidation purposes. Adding full MMU support to Pydgin unfortunately would slow the performance down significantly. As a solution to this problem, Pydgin can directly map guest pages to host pages as long as page sizes align. The page permissions could properly be set so that self modifying code can be efficiently executed. This approach is used by QEMU [Bel05].

Unify gem5 architectural descriptions in PydginFF – While PydginFF, presented in Chapter 4, can allow fast, accurate, and agile cycle-level simulation, one drawback is the multiple architectural descriptions. gem5 has its own DSL for describing instruction sets, while Pydgin uses a DSL embedded in the RPython language. This unfortunately creates two separate places where changes would need to be implemented. Future work could focus on unifying these architectural descriptions. One approach would be to implement a translation pass that converts gem5's descriptions into Pydgin's representation. Similarly, an opposite tool could be implemented to convert Pydgin's representation into gem5's. Another approach would be to use Pydgin for all functional-level modeling, even during detailed simulation. The idea is to deeply integrate Pydgin into gem5, and have Pydgin drive various aspects of cycle-level components (e.g, memories, networks, etc.).

Support for SimJIT in Mamba for longer simulations – Mamba, presented in Chapter 5, demonstrates that meta-tracing JIT VMs can speed up RTL simulations, approaching the performance of commercial Verilog simulators. However, Verilator can be an order of magnitude faster in peak performance than commercial Verilog simulators, and hence from Mamba. Verilator has a long compilation process, which hurts the performance of short-running simulations. On the other hand, the Mamba techniques are compelling for simulations that require fast warmup. Future work for Mamba could combine the best of both worlds: use the Mamba techniques initially, and if the simulation time passes a certain threshold, use PyMTL's SimJIT technique of translating the simulation to Verilog and use Verilator for the rest of simulation [LZB14].

Hybrid event-driven and static scheduling support in Mamba – A similar performance discrepancy in Mamba is seen in activity factors. Static scheduling is a key enabler of performance improvements in Mamba using meta-tracing JIT VMs. Static schedules are significantly more amenable for tracing JITs for finding longer traces that can be optimized well. However, static scheduling has the drawback of simulating every component regardless of the amount of activity in those modules. For modules that are active only in a small percentage of the time still currently needs to be redundantly simulated every cycle. Event-driven scheduling can eliminate many of these redundantly performed combinational update block simulations. Future work might investigate the break-even point where event-driven simulations can outperform static scheduling, and switch to an event-driven simulation methodology when the simulated module has low activity. Furthermore, a hybrid of event-driven and static scheduling could be explored, where a number of combinational update blocks are statically scheduled to superblocks, and each superblock is simulated in an event-driven fashion.

Predication support in RPython for Mamba performance – A meta-tracing JIT compiler extracts a linear trace through the execution of the interpreter. Any control flow in the interpreter, including those due to control flow in the application level, will have only one branch outcome recorded in the trace. If the alternative control flow path gets taken, another trace would be compiled and attached to the failing guard. These additional traces can significantly hurt the memory usage and simulation time. Static scheduling tries to form long traces of execution to allow escape analysis to optimize away intermediate values. However, the longer the trace is, the more bridges would have to be compiled, hurting the execution time and memory usage. Predication support at the meta-tracing level can be very helpful to reduce the number of bridges. Future work can investigate introducing predication IR nodes, and expose these to the application level, as an additional HGSF-aware JIT technique. Many of the control flow operations in the combinational update blocks can be converted to predication operations, and can further improve the performance of Mamba.

Efficient trace dispatch support in RPython for Mamba performance – When the application code contains a branch with many different branch outcomes (e.g., a long `if`, `elif`, `else` chain), RPython compiles a separate trace for each outcome. However, to dispatch to the correct trace, a linear search is performed by failing guards and jumping to an alternative bridge until the bridge with the correct outcome is found. Event-driven RTL simulations typically make use of this pattern heavily, and this hurts the performance of applications that exhibit this branch pattern. This was one of the reasons why static scheduling performed much better than event-driven in Mamba. Future work might explore changes in the RPython framework to perform the trace dispatch more efficiently. For example, a hash map can be used to find the correct bridge, which would have an amortized constant time overhead as opposed to the linear time search.

Combine Pydgin with PyMTL – The first part of this thesis showed that meta-tracing JIT VMs can be useful in accelerating agile hardware simulations across FL, CL, and RTL models. However, the Pydgin and Mamba codebases are currently separate, so they cannot be used together. A unified framework can allow high performance mixed-level modeling across all three of FL, CL, and RTL models. This approach can furthermore allow sampling-based RTL simulation and fast forwarding using a very fast functional model. Unifying PyMTL and Pydgin is a future work.

Hardware-assisted object allocation removal – The software/hardware co-design technique presented in Chapter 7 proposed techniques to optimize away dereferences that have value locality and other pure operations that depend on these dereferences. The mechanism proposed skips over the sub-traces that contain object dereferences and uses the cached value. Sub-trace skipping is only possible if there have not been any writes to the addresses that skipped object dereference would have loaded from. A similar mechanism could be employed to skip sub-traces with new object allocations. Many newly created objects are likely only read within a trace, and never accessed outside the trace. These objects can be reused without the need for creating a new one every iteration of the trace. Future work might explore repurposing the object dereference skipping mechanisms for new object reuse. To skip sub-traces with allocations, the invalidation engine would need to check for read addresses instead of writes, and ensure that the sub-trace is not skipped in case there has been any reads to that new object.

General purpose hardware mechanisms to exploit object dereference locality – One drawback of the proposed hardware mechanisms to exploit object dereference locality is that the additional hardware and instructions might be too specific for the proposed technique. While dynamic languages are an important workload, and meta-tracing JIT VMs easily allow multiple dynamic languages to take advantage of the hardware techniques, more general-purpose hardware techniques that can accelerate additional workloads would make the proposed hardware more compelling. Future work might explore techniques that require less extensive changes in the hardware, reuse more of the commonly found hardware features, or propose hardware additions that can be useful for workloads beyond dynamic languages, while achieving similar benefits of skipping object dereferences with locality.

Hardware support for JIT parameter sweep – While exploring different avenues of building hardware accelerators for meta-tracing JIT VMs, I have observed that some benchmarks have high degrees of sensitivity to some of the meta-tracing JIT parameters. These parameters allow setting various thresholds including the number of iterations a loop has to execute to start tracing, the maximum allowed size of traces, the number of times a guard fails to start compiling a bridge, which optimization passes are allowed, and so on. In particular, we observed that certain benchmarks that have complicated control flows can be significantly affected by the parameter that controls the bridge compilation threshold, with a performance difference more than $2\times$. The stochastic nature of which traces get compiled is a known problem with tracing JIT compilers. Future work might explore hardware techniques to address this variation in performance using auto-tuning or by executing traces compiled with different parameters in parallel, and picking the best performing one.

Hardware support for tracing – In addition to allowing parameter sweeps using hardware mentioned above, there could be additional hardware mechanisms to control tracing in a more systematic way. The stochastic nature of tracing means that pathological traces are very possible. One approach to address this issue could be to let the VM collect multiple traces and pick the most efficient and representative of the executed code. Unfortunately, Chapter 6 shows that the overhead of meta-tracing can be around $100\times$ compared to an interpreter that does not perform tracing, a significant cost. Future work might explore mechanisms for hardware-accelerated tracing that lowers the cost of tracing. If hardware-assisted tracing lowers this overhead, mechanisms to collect multiple traces and compile most frequent and lest pathological ones can make tracing much less stochastic and higher performance. This idea is similar to the one presented by Shull et al. [SCGT19].

Speculative addition of interpreter-level JIT hints – As Chapter 2 shows, the RPython framework provides many interpreter-level JIT hints: constant promotion, elidable functions, quasi-immutables, and virtualizables. Many of these hints express typical access or value patterns as predicted by the interpreter writer. For example, constant promotion is used to treat variables that are typically constant at a particular point in the trace. Using this hint, the JIT compiler inserts a guard to check if the value is indeed what was recorded in the trace, and specializes the rest of the trace for the variable being equal to the recorded constant value. Using these hints requires deep knowledge of tracing, interpreter, and application behavior, and requires the interpreter writer to make assumptions about common application patterns. However, relying on the interpreter writer could be limiting: figuring out the most efficient hints make it less productive to use meta-tracing for new languages, and incorrect or insufficient usage of these hints can hurt performance. Furthermore, different application phases might perform better with different hints. Future work might explore hardware support to speculatively adding and removing some of these hints and observing the performance at each site. Each trace can be recompiled with a slightly different hint. These options can be executed in parallel, and the best performing one can be chosen for the subsequent usage.

Efficient deoptimizations using hardware transactional memory – Chapter 6 shows that deoptimizations can be a big source of overhead. Since the stack layouts of an interpreter and JIT-compiled code are different, the deoptimization step is necessary to safely fall back to the interpreter. While method JITs typically use on-stack replacement, RPython uses a special interpreter that executes the JIT IR operations until a safe point is reached to transfer back to the interpreter. Unfortunately, the performance of this interpreter can be a bottleneck. Future work may explore different strategies of deoptimizing without the need for a JIT IR interpreter. One idea is to use hardware transactional memory to execute the JIT-compiled code between the safe points. If there is any guard failure that necessitates falling back to the interpreter, the work done since the last safe point can be thrown away, and the execution can cheaply transfer back to the interpreter at the last committed safe point. Since this approach throws away useful work, a challenge would be to ensure the overhead of this is less than the current interpreter-based deoptimization approach.

BIBLIOGRAPHY

[AACM07] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. *Symp. on Dynamic Languages*, Oct 2007.

[ABvK$^+$11] O. Almer, I. Böhm, T. E. von Koch, B. Franke, S. Kyle, V. Seeker, C. Thompson, and N. Topham. Scalable Multi-Core Simulation Using Parallel Dynamic Binary Translation. *Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Jul 2011.

[AFCE11] O. Anderson, E. Fortuna, L. Ceze, and S. Eggers. Checked Load: Architectural Support for JavaScript Type-Checking on Mobile Processors. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2011.

[AFF$^+$09] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. COTSon: Infrastructure for Full System Simulation. *ACM SIGOPS Operating Systems Review*, 43(1):52–61, Jan 2009.

[AFG$^+$05] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proc. of the IEEE*, 93(2):449–466, Feb 2005.

[and] ART and Dalvik. Online Webpage. Accessed Dec 25, 2018. `https://source.andoid.com/devices/tech/dalvik`.

[AP14] K. Asanovic and D. A. Patterson. Instruction Sets Should Be Free: The Case for RISC-V. Technical report, UCB/EECS-2014-146, Aug 2014.

[AR13] E. K. Ardestani and J. Renau. ESESC: A Fast Multicore Simulator Using Time-Based Sampling. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2013.

[ARB$^+$05] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros. The ArchC Architecture Description Language and Tools. *Int'l Journal of Parallel Programming (IJPP)*, 33(5):453–484, Oct 2005.

[Ayc03] J. Aycock. A Brief History of Just-in-Time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, Jun 2003.

[BBB$^+$11] N. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 Simulator. *SIGARCH Computer Architecture News (CAN)*, 39(2):1–7, Aug 2011.

[BBH$^+$15] S. Bauman, C. F. Bolz, R. Hirschfeld, V. Kirilichev, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. Pycket: A Tracing JIT for a Functional Language. *ACM SIGPLAN Int'l Conf. on Functional Programming*, Aug 2015.

[BBT13] E. Barrett, C. F. Bolz, and L. Tratt. Unipycation: A Case Study in Cross-Language Tracing. *ACM Workshop on Virtual Machines and Intermediate Languages*, Oct 2013.

[BCF+11a] C. F. Bolz, A. Cuni, M. Fijalkowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation Removal by Partial Evaluation in a Tracing JIT. *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, Jan 2011.

[BCF+11b] C. F. Bolz, A. Cuni, M. Fijalkowski, S. Pedroni, and A. Rigo. Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages. *Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*, Jul 2011.

[BCFR09] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. *Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*, Jul 2009.

[BCSS98] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. *Int'l Conf. on Functional Programming (ICFP)*, Sep 1998.

[BDM+07] S. Belloeil, D. Dupuis, C. Masson, J. Chaput, and H. Mehrez. Stratus: A Procedural Circuit Description Language Based Upon Python. *Int'l Conf. on Microelectronics (ICM)*, Dec 2007.

[Bea10] D. Beazley. Understanding the Python GIL. *PyCon*, Feb 2010.

[Bed04] R. Bedicheck. SimNow: Fast Platform Simulation Purely in Software. *Symp. on High Performance Chips (Hot Chips)*, Aug 2004.

[BEK07] F. Brandner, D. Ebner, and A. Krall. Compiler Generation from Structural Architecture Descriptions. *Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Sep 2007.

[Bel05] F. Bellard. QEMU, A Fast and Portable Dynamic Translator. *USENIX Annual Technical Conference (ATEC)*, Apr 2005.

[BFKR09] F. Brandner, A. Fellnhofer, A. Krall, and D. Riegler. Fast and Accurate Simulation using the LLVM Compiler Framework. *Workshop on Rapid Simulation and Performance Evalution: Methods and Tools (RAPIDO)*, Jan 2009.

[BFT10] I. Böhm, B. Franke, and N. Topham. Cycle-Accurate Performance Modelling in an Ultra-Fast Just-In-Time Dynamic Binary Translation Instruction Set Simulator. *Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Jul 2010.

[BFT11] I. Böhm, B. Franke, and N. Topham. Generalized Just-In-Time Trace Compilation Using a Parallel Task Farm in a Dynamic Binary Translator. *Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2011.

[BH98] P. Bellows and B. Hutchings. JHDL-An HDL for Reconfigurable Systems. *Symp. on FPGAs for Custom Computing Machines (FCCM)*, Apr 1998.

[BHJ$^+$11] F. Blanqui, C. Helmstetter, V. Jolobok, J.-F. Monin, and X. Shi. Designing a CPU Model: From a Pseudo-formal Document to Fast Code. *CoRR arXiv:1109.4351*, Sep 2011.

[BKK$^+$10] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. Cλash: Structural Descriptions of Synchronous Hardware Using Haskell. *Euromicro Conf. on Digital System Design (DSD)*, Sep 2010.

[BKL$^+$08] C. F. Bolz, A. Kuhn, A. Lienhard, N. D. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, and T. Verwaest. Back to the Future in One Week — Implementing a Smalltalk VM in PyPy. *Workshop on Self-Sustaining Systems (S3)*, May 2008.

[Blo70] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, Jul 1970.

[BLS10] C. F. Bolz, M. Leuschel, and D. Schneider. Towards a Jitting VM for Prolog Execution. *Int'l Symp. on Principles and Practice of Declarative Programming (PPDP)*, Jul 2010.

[BNH$^+$04] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, Jun 2004.

[Bol12] C. F. Bolz. *Meta-Tracing Just-In-Time Compilation for RPython*. Ph.D. Thesis, Mathematisch-Naturwissenschaftliche Fakultät, Heinrich Heine Universität Düsseldorf, 2012.

[BPSTH14] C. F. Bolz, T. Pape, J. Siek, and S. Tobin-Hochstadt. Meta-Tracing Makes a Fast Racket. *Workshop on Dynamic Languages and Applications (DYLA)*, Jun 2014.

[Bro76] P. J. Brown. Throw-Away Compiling. *Journal of Software – Practice and Experience*, 6(3):423–434, Jul 1976.

[Bru09] S. Brunthaler. Virtual-Machine Abstraction and Optimization Techniques. *Electronic Notes in Theoretical Computer Science*, Dec 2009.

[BSGG14] A. Branković, K. Stavrou, E. Gibert, and A. González. Warm-Up Simulation Methodology for HW/SW Co-Designed Processors. *Int'l Symp. on Code Generation and Optimization (CGO)*, Feb 2014.

[BT15] C. F. Bolz and L. Tratt. The Impact of Meta-Tracing on VM Design and Implementation. *Science of Computer Programming*, 98(3):408–421, Feb 2015.

[BV09] C. Bruni and T. Verwaest. PyGirl: Generating Whole-System VMs from High-Level Prototypes Using PyPy. *Int'l Conf. on Objects, Components, Models, and Patterns (TOOLS-EUROPE)*, Jun 2009.

[BVR$^+$12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. *Design Automation Conf. (DAC)*, Jun 2012.

[Car18] P. Carbonnelle. PYPL PopularitY of Programming Language. Online Webpage, 2018. Accessed Dec 24, 2018.
`http://pypl.github.io/PYPL.html`.

[Cas18] S. Cass. IEEE Spectrum The 2018 Top Programming Languages. Online Webpage, 2018. Accessed Dec 24, 2018.
`https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages`.

[CDS$^+$14] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Feb 2014.

[CEI$^+$12] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu. On the Benefits and Pitfalls of Extending a Statically Typed Language JIT Compiler for Dynamic Scripting Languages. *ACM SIGPLAN Notices*, 47(10):195–212, Oct 2012.

[cha] ChakraCore. Online Webpage. Accessed Dec 25, 2018.
`https://github.com/Microsoft/ChakraCore`.

[CHE11] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*, Nov 2011.

[CHL$^+$04] J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. Modeling Instruction Semantics in ADL Processor Descriptions for C Compiler Retargeting. *Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Jul 2004.

[CHL$^+$05] J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. C Compiler Retargeting Based on Instruction Semantics Models. *Design, Automation, and Test in Europe (DATE)*, Mar 2005.

[CHM96] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing State Loss for Effective Trace Sampling of Superscalar Processors. *Int'l Conf. on Computer Design (ICCD)*, Oct 1996.

[CK94] B. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, May 1994.

[CLN$^+$11] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 30(4):473–491, Mar 2011.

[CSGT17] J. Choi, T. Shull, M. J. Garzaran, and J. Torrellas. ShortCut: Architectural Support for Fast Object Access in Scripting Languages. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[CTD$^+$17] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood. A Pythonic Approach for Rapid Hardware Prototyping and Instrumentation. *Int'l Conf. on Field Programmable Logic (FPL)*, Sep 2017.

[CTW05] C. Click, G. Tene, and M. Wolf. The Pauseless GC Algorithm. *ACM/USENIX Int'l Conf. on Virtual Execution Environments (VEE)*, Jun 2005.

[CU91] C. Chambers and D. Ungar. Making Pure Object-Oriented Langauges Practical. *Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, Oct 1991.

[CWZ90] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of Pointers and Structures. *Conf. on Programming Language Design and Implementation (PLDI)*, Jun 1990.

[DA99] D. Detlefs and O. Agesen. The Case for Multiple Compilers. *Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, Nov 1999.

[Daw73] J. L. Dawson. Combining Interpretive Code with Machine Code. *The Computer Journal*, 16(3):216–219, Jan 1973.

[Dec04] J. Decaluwe. MyHDL: A Python-based Hardware Description Language. *Linux Journal*, Nov 2004.

[DGY$^+$74] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits (JSSC)*, 9(5):256–268, Oct 1974.

[DMG15] G. Dot, A. Martínez, and A. González. Analysis and Optimization of Engines for Dynamically Typed Languages. *Int'l Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct 2015.

[DMG17] G. Dot, A. Martínez, and A. González. Removing Checks in Dynamically Typed Languages through Efficient Profiling. *Int'l Symp. on Code Generation and Optimization (CGO)*, Feb 2017.

[DMM98] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-Based Alias Analysis. *Conf. on Programming Language Design and Implementation (PLDI)*, Jun 1998.

[DP73] R. J. Dakin and P. C. Poole. A Mixed Code Approach. *The Computer Journal*, 16(3):219–222, Jan 1973.

[DQ06] J. D'Errico and W. Qin. Constructing Portable Compiled Instruction-Set Simulators — An ADL-Driven Approach. *Design, Automation, and Test in Europe (DATE)*, Mar 2006.

[DS84] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. *ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL)*, Jan 1984.

[DSF09] M. T. Daly, V. Sazawal, and J. S. Foster. Work In Progress: An Empirical Study of Static Typing in Ruby. *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, Oct 2009.

[EG01] M. Ertl and D. Gregg. The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures. *Euro-Par 2001*, Aug 2001.

[ELDJ05] L. Eeckhout, Y. Luo, K. De Bosschere, and L. K. John. BLRL: Accurate and Efficient Warmup for Sampled Processor Simulation. *The Computer Journal*, May 2005.

[FKSB06] S. Farfeleder, A. Krall, E. Steiner, and F. Brandner. Effective Compiler Generation by Architecture Description. *International Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES)*, Jun 2006.

[FMP13] N. Fournel, L. Michel, and F. Pétrot. Automated Generation of Efficient Instruction Decoders for Instruction Set Simulators. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2013.

[Fog18] A. Fog. Instruction Tables. Online Webpage, Sep 2018. `http://www.agner.org`.

[Fra08] B. Franke. Fast Cycle-Approximate Instruction Set Simulation. *Workshop on Software & Compilers for Embedded Systems (SCOPES)*, Mar 2008.

[Fut71] Y. Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[gcj18] Guide to GNU gcj. Online Webpage, 2018. Accessed Dec 25, 2018. `https://gcc.gnu.org/onlinedocs/gcc-6.5.0/gcj/`.

[GES$^+$09] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based Just-in-Time Type Specialization for Dynamic Languages. *Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2009.

[GM98] F. Gabbay and A. Mendelson. Using Value Prediction to Increase the Power of Speculative Execution Hardware. *ACM Trans. on Computer Systems (TOCS)*, 16(3):234–270, Aug 1998.

[Gou] I. Gouy. The Computer Language Benchmarks Game. `http://benchmarksgame.alioth.debian.org/`.

[GSL17] D. Gope, D. J. Schalis, and M. H. Lipasti. Architectural Support for Server-Side PHP Processing. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[GTBS13] J. P. Grossman, B. Towles, J. A. Bank, and D. E. Shaw. The Role of Cascade, a Cycle-Based Simulation Infrastructure, in Designing the Anton Special-Purpose Supercomputers. *Design Automation Conf. (DAC)*, Jun 2013.

[GTM99] A. Gonzalez, J. Tubella, and C. Molina. Trace-Level Reuse. *Int'l Conf. on Parallel Processing (ICPP)*, Sep 1999.

[HCU91] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. *European Conference on Object-Oriented Programming (ECOOP)*, Jul 1991.

[HCU92] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. *Conf. on Programming Language Design and Implementation (PLDI)*, Jun 1992.

[HH09] A. Holkner and J. Harland. Evaluating the Dynamic Behaviour of Python Applications. *Australasian Conf. on Computer Science*, Jan 2009.

[HHR95] R. E. Hank, W.-M. W. Hwu, and B. R. Rau. Region-Based Compilation: An Introduction and Motivation. *Int'l Symp. on Microarchitecture (MICRO)*, Nov 1995.

[hip15] HippyVM PHP Implementation. Online Webpage, 2014 (accessed Jan 14, 2015). `http://www.hippyvm.com`.

[HKR$^+$14] S. Hanenberg, S. Kelinschmanger, R. Robbes, E. Tater, and A. Stefik. An Empirical Study on the Impact of Static Typing on Software Maintainability. *Empirical Software Engineering*, 19(5):1335–1382, Oct 2014.

[HL99] J. Huang and D. J. Lilja. Exploiting Basic Block Value Locality with Block Reuse. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Jan 1999.

[HMLT03] P. Haglund, O. Mencer, W. Luk, and B. Tai. Hardware Design with a Scripting Language. *Int'l Conf. on Field Programmable Logic (FPL)*, Sep 2003.

[Höl13] M. Hölttä. Crankshafting from the Ground Up. *Google Technical Report*, Aug 2013.

[HS05] J. W. Haskins Jr. and K. Skadron. Accelerated Warmup for Sampled Microarchitecture Simulation. *ACM Trans. on Architecture and Code Optimization (TACO)*, Mar 2005.

[HSK$^+$04] M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and H. van Someren. A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models. *Design, Automation, and Test in Europe (DATE)*, Feb 2004.

[HTCU10] T. Harris, S. Tomic, A. Cristal, and O. Unsal. Dynamic Filtering: Multi-Purpose Architecture Support for Language Runtime Systems. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2010.

[HU94a] U. Hölzle and D. Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. *Conf. on Programming Language Design and Implementation (PLDI)*, Jun 1994.

[HU94b] U. Hölzle and D. Ungar. A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance. *Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, Oct 1994.

[IB16] B. Ilbeyi and C. Batten. JIT-Assisted Fast-Forward Embedding and Instrumentation to Enable Fast, Accurate, and Agile Simulation. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2016.

[IBTB17] B. Ilbeyi, C. F. Bolz-Tereick, and C. Batten. Cross-Layer Workload Characterization of Meta-Tracing JIT VMs. *Int'l Symp. on Workload Characterization (IISWC)*, Oct 2017.

[ica] Icarus Verilog. `http://iverilog.icarus.com`.

[ILB16] B. Ilbeyi, D. Lockhart, and C. Batten. Pydgin for RISC-V: A Fast and Productive Instruction-Set Simulator. *RISC-V Workshop*, Apr 2016.

[IS18] M. Ismail and G. E. Suh. Quantitative Overhead Analysis for Python. *Int'l Symp. on Workload Characterization (IISWC)*, Sep 2018.

[jav] JavaScriptCore. Online Webpage. Accessed Dec 25, 2018. `https://trac.webkit.org/wiki/JavaScriptCore`.

[JB99] J. Jennings and E. Beuscher. Verischemelog: Verilog Embedded in Scheme. *Conf. on Domain-Specific Languages (DSL)*, Oct 1999.

[JCLJ08] A. Jaleel, R. Cohn, C. Luk, and B. Jacob. A Pin-Based On-the-Fly Multi-Core Cache Simulator. *Workshop on Modeling, Benchmarking and Simulation (MOBS)*, Jun 2008.

[JG02] N. D. Jones and A. J. Glenstrup. Program Generation, Termination, and Binding-Time Analysis. *Int'l Conf. on Generative Programming and Component Engineering*, Oct 2002.

[JIB18] S. Jiang, B. Ilbeyi, and C. Batten. Mamba: Closing the Performance Gap in Productive Hardware Development Frameworks. *Design Automation Conf. (DAC)*, Jun 2018.

[JL96] R. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Sep 1996.

[JMP09] J. A. Joao, O. Mutlu, and Y. N. Patt. Flexible Reference-Counting-Based Hardware Acceleration for Garbage Collection. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2009.

[JT09] D. Jones and N. Topham. High Speed CPU Simulation Using LTU Dynamic Binary Translation. *Int'l Conf. on High Performance Embedded Architectures and Compilers (HiPEAC)*, Jan 2009.

[jul] The Julia Programming Language. Online Webpage. Accessed Dec 25, 2018. `http://julialang.org`.

[JYP$^+$17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[KA01] R. Krishna and T. Austin. Efficient Software Decoder Design. *Workshop on Binary Translation (WBT)*, Sep 2001.

[KBF$^+$12] S. Kyle, I. Böhm, B. Franke, H. Leather, and N. Topham. Efficiently Parallelizing Instruction Set Simulation of Embedded Multi-Core Processors Using Region-Based Just-In-Time Dynamic Binary Translation. *International Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES)*, Jun 2012.

[KFML00] A. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research. *Int'l Conf. on Computer Design (ICCD)*, Sep 2000.

[KKK+17] C. Kim, J. Kim, S. Kim, D. Kim, N. Kim, G. Na, Y. H. Oh, H. G. Cho, and J. W. Lee. Typed Architectures: Architectural Support for Lightweight Scripting. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr 2017.

[KM05] T. Kotzmann and H. Mössenböck. Escape Analysis in the Context of Dynamic Compilation and Deoptimization. *ACM/USENIX Int'l Conf. on Virtual Execution Environments (VEE)*, Jun 2005.

[Koz18] V. Kozlov. JEP 295: Ahead-of-Time Compilation. Online Webpage, 2018. Accessed Dec 25, 2018.
`https://openjdk.java.net/jeps/295`.

[Lan13] LangPop. Programming Language Popularity. Online Webpage, 2013. Accessed Dec 24, 2018.
`http://65.39.133.14/`.

[LB09] M. J. Lyons and D. Brooks. The Design of a Bloom Filter Hardware Accelerator for Ultra Low Power Systems. *Int'l Symp. on Low-Power Electronics and Design (ISLPED)*, Aug 2009.

[LCL+11] Y. Lifshitz, R. Cohn, I. Livni, O. Tabach, M. Charney, and K. Hazelwood. Zsim: A Fast Architectural Simulator for ISA Design-Space Exploration. *Workshop on Infrastructures for Software/Hardware Co-Design (WISH)*, Apr 2011.

[LCM+05] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2005.

[LIB15] D. Lockhart, B. Ilbeyi, and C. Batten. Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar 2015.

[LL00] Y. Li and M. Leeser. HML, A Novel Hardware Description Language and Its Translation to VHDL. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, 8(1):1–8, Dec 2000.

[Loc15] D. Lockhart. *Constructing Vertically Integrated Hardware Design Methodologies Using Embedded Domain-Specific Languages and Just-in-Time Optimization*. Ph.D. Thesis, Cornell University, 2015.

[LPC06] J. Lau, E. Perelman, and B. Calder. Selecting Software Phase Markers with Code Structure Analysis. *Int'l Symp. on Code Generation and Optimization (CGO)*, Mar 2006.

[LPP88] S. Laha, J. H. Patel, and R. K. Patel. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Trans. on Computers (TC)*, Nov 1988.

[LS00] T. Lafage and A. Seznec. Choosing Representative Slices of Program Execution for Microarchitecture Simulations: A Preliminary Application to the Data Stream. *IEEE Annual Workshop on Workload Characterization*, Sep 2000.

[LSC04] J. Lau, S. Schoenmackers, and B. Calder. Structures for Phase Classification. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar 2004.

[lua] The LuaJIT Project. Online Webpage. Accessed Dec 25, 2018. `http://luajit.org`.

[LZB14] D. Lockhart, G. Zibrat, and C. Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[MAK18] M. Maas, K. Asanović, and J. Kubiatowicz. A Hardware Accelerator for Tracing Garbage Collection. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2018.

[Mas07] A. Mashtizadeh. *PHDL: A Python Hardware Design Framework.* M.S. Thesis, EECS Department, MIT, May 2007.

[May87] C. May. Mimic: A Fast System/370 Simulator. *ACM Sigplan Symp. on Interpreters and Interpretive Techniques*, Jun 1987.

[MBDH99] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A Portable Interface to Hardware Performance Counters. *DoD HPCMP Users Gp. Conf.*, Jun 1999.

[MD15] S. Marr and S. Ducasse. Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approachess for Self-Optimizing Interpreters. *Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, Dec 2015.

[Mey06] M. Meyer. A True Hardware Read Barrier. *Int'l Symp. on Memory Management (ISMM)*, Jun 2006.

[mig] Migen. `https://m-labs.hk/gateware.html`.

[MKK$^+$10] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Jan 2010.

[MO98] H. McGhan and M. O'Connor. PicoJava: A Direct Execution Engine for Java Bytecode. *IEEE Computer*, 31(10):22–30, Oct 1998.

[Moo65] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics Magazine*, 1965.

[MRF$^+$00] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. Huss-Lederman, and J. R. Larus. Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator. *IEEE Concurrency*, 1(4):12–20, Oct 2000.

[MSB$^+$05] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. M. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution Driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News (CAN)*, 33(4):92–99, Sep 2005.

[MZ04] W. S. Mong and J. Zhu. DynamoSim: A Trace-Based Dynamically Compiled Instruction Set Simulator. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2004.

[NBS$^+$02] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. *Design Automation Conf. (DAC)*, Jun 2002.

[O'G18] S. O'Grady. The RedMonk Programming Language Rankings: June 2018. Online Webpage, 2018. Accessed Dec 24, 2018.
`https://redmonk.com/sogrady/2018/08/10/language-rankings-6-18/`.

[Ott18] G. Ottorni. HHVM JIT: A Profile-Guided, Region-Based Compiler for PHP and Hack. *Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2018.

[PACG11] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS: A Full System Simulator for Multicore x86 CPUs. *Design Automation Conf. (DAC)*, Jun 2011.

[PC11] D. A. Penry and K. D. Cahill. ADL-Based Specification of Implementation Styles for Functional Simulators. *Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Jul 2011.

[PCC$^+$04] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2004.

[Pen11] D. A. Penry. A Single-Specification Principle for Functional-to-Timing Simulator Interface Design. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2011.

[Pet08] B. Peterson. PyPy. In A. Brown and G. Wilson, editors, *The Architecture of Open Source Applications, Volume II*. LuLu.com, 2008.

[PHC03] E. Perelman, G. Hamerly, and B. Calder. Picking Statisticlally Valid and Early Simulation Points. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2003.

[pin15] Pin–A Dynamic Binary Instrumentation Tool. Online Webpage, 2012 (accessed Sep, 2015). `http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.

[PKHK11] Z. Přikryl, J. Křoustek, T. Hruška, and D. Kolář. Fast Just-In-Time Translated Simulator for ASIP Design. *Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, Apr 2011.

[PLP$^+$07] E. Perelman, J. Lau, H. Patil, A. Jaleel, G. Harmerly, and B. Calder. Cross Binary Simulation Points. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2007.

[PMT04] D. G. Pérez, G. Mouchard, and O. Temam. A New Optimized Implemention of the SystemC Engine Using Acyclic Scheduling. *Design, Automation, and Test in Europe (DATE)*, Feb 2004.

[Por05] C. Porthouse. Jazelle for Execution Environments. ARM Whitepaper, 2005.

[pyd15] Pydgin Repository on GitHub. Online Webpage, 2015 (accessed Sep 15, 2015). `http://www.github.com/cornell-brg/pydgin`.

[pypa] PyPy. Online Webpage. (accessed Dec 25, 2018) `http://www.pypy.org`.

[pypb] PyPy Benchmark Suite. `https://bitbucket.org/pypy/benchmarks`.

[QDZ06] W. Qin, J. D'Errico, and X. Zhu. A Multiprocessing Approach to Accelerate Retargetable and Portable Dynamic-Compiled Instruction-Set Simulation. *Intl'l Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, Oct 2006.

[QM03a] W. Qin and S. Malik. Automated Synthesis of Efficient Binary Decoders for Retargetable Software Toolkits. *Design Automation Conf. (DAC)*, Jun 2003.

[QM03b] W. Qin and S. Malik. Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation. *Design, Automation, and Test in Europe (DATE)*, Jun 2003.

[QM05] W. Qin and S. Malik. A Study of Architecture Description Languages from a Model-based Perspective. *Workshop on Microprocessor Test and Verification (MTV)*, Nov 2005.

[QRM04] W. Qin, S. Rajagopalan, and S. Malik. A Formal Concurrency Model Based Architecture Description Language for Synthesis of Software Development Tools. *International Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES)*, Jun 2004.

[RABA04] S. Rigo, G. Araújo, M. Bartholomeu, and R. Azevedo. ArchC: A SystemC-Based Architecture Description Language. *Int'l Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct 2004.

[Rau78] B. R. Rau. Levels of Representation of Programs and the Architecture of Universal Host Machines. *Annual Workshop on Microprogramming (MICRO)*, Nov 1978.

[RBMD03] M. Reshadi, N. Bansal, P. Mishra, and N. Dutt. An Efficient Retargetable Framework for Instruction-Set Simulation. *Intl'l Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, Oct 2003.

[RD03] M. Reshadi and N. Dutt. Reducing Compilation Time Overhead in Compiled Simulators. *Int'l Conf. on Computer Design (ICCD)*, Oct 2003.

[RDM06] M. Reshadi, N. Dutt, and P. Mishra. A Retargetable Framework for Instruction-Set Architecture Simulation. *IEEE Trans. on Embedded Computing Systems (TECS)*, May 2006.

[RGM16] M. Rigger, M. Grimmer, and H. Mössenböck. Sulong – Execution of LLVM-Based Languages on the JVM. *Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*, Jul 2016.

[RHL$^+$93] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, May 1993.

[RLBV10] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. *Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2010.

[RLZ10] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. *WebApps 2010*, Jun 2010.

[RMD03] M. Reshadi, P. Mishra, and N. Dutt. Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation. *Design Automation Conf. (DAC)*, Jun 2003.

[RMD09] M. Reshadi, P. Mishra, and N. Dutt. Hybrid-Compiled Simulation: An Efficient Technique for Instruction-Set Architecture Simulation. *IEEE Trans. on Embedded Computing Systems (TECS)*, Apr 2009.

[RO14] J. M. Redondo and F. Ortin. A Comprehensive Evaluation of Common Python Programs. *IEEE Software*, 32(4):76–84, Jul 2014.

[RP06] A. Rigo and S. Pedroni. PyPy's Approach to Virtual Machine Construction. *Symp. on Dynamic Languages*, Oct 2006.

[RPOM05] J. Ringenberg, C. Pelosi, D. Oehmke, and T. Mudge. Intrinsic Checkpointing: A Methodology for Decreasing Simulation Time Through Binary Modification. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar 2005.

[RSS15] E. Rohou, B. N. Swamy, and A. Seznec. Branch Prediction and the Performance of Interpreters: Don't Trust Folklore. *Int'l Symp. on Code Generation and Optimization (CGO)*, Feb 2015.

[SAMC99] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch Prediction, Instruction-Window Size, and Cache Size: Performance Tradeoffs and Simulation Techniques. *IEEE Trans. on Computers (TC)*, Nov 1999.

[SAW$^+$10] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. Stevenson, A. Solomatnikov, A. Firoozshahian, B. Lee, S. Richardson, and M. Horowitz. Rethinking Digital Design: Why Design Must Change. *IEEE Micro*, 30(6):9–24, Nov/Dec 2010.

[SB12] D. Schneider and C. F. Bolz. The Efficient Handling of Guards in the Design of RPython's Tracing JIT. *ACM Workshop on Virtual Machines and Intermediate Languages*, Oct 2012.

[SBC00] V. C. Sreedhar, M. Burke, and J.-D. Choi. A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading. *ACM SIGPLAN Notices*, 35(5):196–207, Aug 2000.

[SCGT19] T. Shull, J. Choi, M. J. Garzaran, and J. Torellas. NoMap: Speeding-Up JavaScript Using Hardware Transactional Memory. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2019.

[Sea15] C. Seaton. *Specialising Dynamic Techniques for Implementing the Ruby Programming Language*. Ph.D. Thesis, School of Computer Science, University of Manchester, 2015.

[SF14] C. Souza and E. Figueiredo. How Do Programmers Use Optional Typing? An Empirical Study. *Int'l Conf. on Modularity*, Apr 2014.

[SK13] D. Sanchez and C. Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2013.

[SL98] E. Schnarr and J. R. Larus. Fast Out-of-Order Processor Simulation Using Memoization. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 1998.

[SMB+04] P. K. Szwed, D. Marques, R. M. Buels, S. A. McKee, and M. Schulz. SimSnap: Fast-Forwarding via Native Execution and Application-Level Checkpointing. *Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*, Feb 2004.

[Smi81] J. E. Smith. A Study of Branch Prediction Strategies. *Int'l Symp. on Computer Architecture (ISCA)*, May 1981.

[SNC+15] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer. Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed. *Int'l Symp. on Workload Characterization (IISWC)*, Oct 2015.

[SPB+13] A. Sarimbekov, A. Pdzimek, L. Bulej, Y. Zheng, N. Ricci, and W. Binder. Characteristics of Dynamic JVM Languages. *ACM Workshop on Virtual Machines and Intermediate Languages*, Oct 2013.

[SPHC02] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Feb 2002.

[spi] SpiderMonkey. Online Webpage. Accessed Dec 25, 2018. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`.

[SR16] G. Southern and J. Renau. Overhead of Deoptimization Checks in the V8 JavaScript Engine. *Int'l Symp. on Workload Characterization (IISWC)*, Sep 2016.

[Sri18] S. Srinath. *Lane-Based Hardware Specialization for Loop- and Fork-Join-Centric Parallelization and Scheduling Strategies*. Ph.D. Thesis, Cornell University, 2018.

[SS95] J. E. Smith and G. S. Sohi. The Microarchitecture of Superscalar Processors. *Proc. of the IEEE*, Dec 1995.

[SS97] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 1997.

[SWHJ16] L. Stadler, A. Welc, C. Humer, and M. Jordan. Optimizing R Language Execution via Aggressive Speculation. *Symp. on Dynamic Languages*, Nov 2016.

[SWM14] L. Stadler, T. Würthinger, and H. Mössenböck. Partial Escape Analysis and Scalar Replacement for Java. *Int'l Symp. on Code Generation and Optimization (CGO)*, Feb 2014.

[TEL95] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 1995.

[TGS18] P.-A. Tsai, Y. L. Gan, and D. Sanchez. Rethinking the Memory Hierarchy for Modern Languages. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2018.

[TIO18] TIOBE. TIOBE Index for December 2018. Online Webpage, 2018. Accessed Dec 24, 2018.
`https://www.tiobe.com/tiobe-index/`.

[TIW11] G. Tene, B. Iyengar, and M. Wolf. C4: The Continuously Concurrent Compacting Collector. *Int'l Symp. on Memory Management (ISMM)*, Jun 2011.

[TJ07] N. Topham and D. Jones. High Speed CPU Simulation using JIT Binary Translation. *Workshop on Modeling, Benchmarking and Simulation (MOBS)*, Jun 2007.

[TJAH$^+$18] C. Torng, S. Jiang, K. Al-Hawaj, I. Bukreyev, B. Ilbeyi, T. Ta, L. Cheng, J. Puscar, I. Galton, and C. Batten. A New Era of Silicon Prototyping in Computer Architecture Research. *RISC-V Day Workshop*, Oct 2018.

[tlp13] The Transparent Language Popularity Index. Online Webpage, 2013. Accessed Dec 24, 2018.
`http://lang-index.sourceforge.net/`.

[top15] Topaz Ruby Implementation. Online Webpage, 2014 (accessed Jan 14, 2015). `http://github.com/topazproject/topaz`.

[Tre15] Trendy Skills Blog. Best Paid Programming Language in 2014. Online Webpage, 2015. Accessed Dec 24, 2018.
`https://trendyskills.com/blog/tag/best-paid-programming-languages/`.

[trua] FastR. `https://github.com/graalvm/truffleruby`.

[trub] TruffleRuby. Online Webpage. Accessed Dec 25, 2018.
`https://github.com/oracle/truffleruby`.

[v8] What is v8? Online Webpage. Accessed Dec 25, 2018.
`https://v8.dev`.

[VCE06] M. Van Biesbrouck, B. Calder, and L. Eeckhout. Efficient Sampling Startup for SimPoint. *IEEE Micro*, Jul 2006.

[ver14] Verilator. Online Webpage, 2014 (accessed Oct 1, 2014). `http://www.veripool.org/wiki/verilator`.

[WA17] A. Waterman and K. Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2. Technical report, RISC-V Foundation, May 2017.

[WB13] C. Wimmer and S. Brunthaler. ZipPy on Truffle: A Fast and Simple Implementation of Python. *Conf. on Systems, Programming, and Application: Software for Humanity (SPLASH)*, Oct 2013.

[WGFT13] H. Wagstaff, M. Gould, B. Franke, and N. Topham. Early Partial Evaluation in a JIT-Compiled, Retargetable Instruction Set Simulator Generated from a High-Level Architecture Description. *Design Automation Conf. (DAC)*, Jun 2013.

[Wil92] P. R. Wilson. Uniprocessor Garbage Collection Techniques. *Int'l Workshop on Memory Management*, Sep 1992.

[WR96] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, May 1996.

[WWFH03] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2003.

[WWFH06a] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Simulation Sampling with Live-Points. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar 2006.

[WWFH06b] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Statistical Sampling of Microarchitecture Simulation. *ACM Trans. on Modeling and Computer Simulation*, Jul 2006.

[WWH$^+$17] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. *Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2017.

[WWS$^+$12] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-Optimizing AST Interpreters. *Symp. on Dynamic Languages*, Oct 2012.

[WWW$^+$13] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. *Int'l Symp. on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, Oct 2013.

[YG00] J. Yang and R. Gupta. Load Redundancy Removal Through Instruction Reuse. *Int'l Conf. on Parallel Processing (ICPP)*, Aug 2000.

[ŽPM96] V. Živojnović, S. Pees, and H. Meyr. LISA - Machine Description Language and Generic Machine Model for HW/SW Co-Design. *Workshop on VLSI Signal Processing*, Oct 1996.