CONSTRUCTING VERTICALLY INTEGRATED HARDWARE DESIGN METHODOLOGIES USING EMBEDDED DOMAIN-SPECIFIC LANGUAGES AND JUST-IN-TIME OPTIMIZATION

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

Derek Matthew Lockhart

August 2015

CONSTRUCTING VERTICALLY INTEGRATED HARDWARE DESIGN METHODOLOGIES
USING EMBEDDED DOMAIN-SPECIFIC LANGUAGES AND JUST-IN-TIME
OPTIMIZATION

Derek Matthew Lockhart, Ph.D.

Cornell University 2015

The growing complexity and heterogeneity of modern application-specific integrated circuits has made hardware design methodologies a limiting factor in the construction of future computing systems. This work aims to alleviate some of these design challenges by embedding productive hardware modeling and design constructs in general-purpose, high-level languages such as Python. Leveraging Python-based embedded domain-specific languages (DSLs) can considerably improve designer productivity over traditional design flows based on hardware-description languages (HDLs) and C++, however, these productivity benefits can be severely impacted by the poor execution performance of Python simulations. To address these performance issues, this work combines Python-based embedded-DSLs with just-in-time (JIT) optimization strategies to generate high-performance simulators that significantly reduce this performance-productivity gap. This thesis discusses two frameworks I have constructed that use this novel design approach: PyMTL, a Python-based, concurrent-structural modeling framework for vertically integrated hardware design, and Pydgin, a framework for generating high-performance, just-in-time optimizing instruction set simulators from high-level architecture descriptions.

BIOGRAPHICAL SKETCH

Derek M. Lockhart was born to Bonnie Lockhart and Scott Lockhart in the suburbs of Saint Louis, Missouri on August 8th, 1983. He grew up near Creve Coeur, Missouri under the guidance of his mother; he spent his summers in Minnesota, California, Florida, Colorado, Texas, New Hampshire, Massachusetts, and Utah to visit his frequently moving father. In high school, Derek dedicated himself to cross country and track in addition to his academic studies. He graduated from Pattonville High School as both a valedictorian and a St. Louis Post-Dispatch Scholar Athlete.

Determined to attend an undergraduate institution as geographically distant from St. Louis as possible, Derek enrolled at the California Polytechnic State University in San Luis Obispo. At Cal Poly, Derek tried his best to be an engaged member of the campus community by serving as President of the Tau Beta Pi engineering honor society, giving university tours as an Engineering Ambassador, and even becoming a member of the Persian Students of Cal Poly club. He completed a degree in Computer Engineering in 2007, graduating Magna Cum Laude and With Honors.

Motivated by his undergraduate research experiences working under Dr. Diana Franklin and by an internship in the Platforms group at Google, Derek decided to pursue a doctorate degree in the field of computer architecture. He chose to trek across the country yet again in order to accept an offer from Cornell University where he was admitted as a Jacobs Fellow. After several years and a few failed research projects, Derek found his way into the newly created lab of Professor Christopher Batten. During his time at Cornell, he received an education in Electrical and Computer Engineering, with a minor in Computer Science.

Derek has recently accepted a position as a hardware engineer in the Platforms group of Google in Mountain View, CA. There he hopes to help build amazing datacenter hardware and continue his quest to create a design methodology that will revolutionize the process of hardware design.

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

MTL	modeling towards layout
FL	functional level
CL	cycle-level
RTL	register-transfer level
GL	gate-level
DSL	domain-specific language
DSEL	domain-specific embedded language
EDSL	embedded domain-specific language
ADL	architectural description language
ELL	efficiency-level language
PLL	performance-level language
HDL	hardware description language
HGL	hardware generation language
HLS	high-level synthesis
CAS	cycle-approximate simulator
ISS	instruction set simulator
VM	virtual machine
WSVM	whole-system virtual machine
VCD	value change dump
FFI	foreign-function interface
CFFI	C foreign-function interface
DBT	dynamic binary translation
JIT	just-in-time compiler
SEJITS	selective embedded just-in-time specialization
IR	intermediate representation
ISA	instruction set architecture
RISC	reduced instruction set computer
CISC	complex instruction set computer
PC	program counter
MIPS	million instructions per second
SRAM	static random access memory
CAD	computer aided design
EDA	electronic-design automation
VLSI	very-large-scale integration
ASIC	application-specific integrated circuit
ASIP	application-specific instruction set processor
FPGA	field-programmable gate array
SOC	system-on-chip
OCN	on-chip network

CHAPTER 1
INTRODUCTION

Since the invention of the transistor in 1947, technology improvements in the manufacture of digital integrated circuits has provided hardware architects with increasingly capable building blocks for constructing digital systems. While these semiconductor devices have always come with limitations and trade-offs with respect to performance, area, power, and energy, computer architects could rely on technology scaling to deliver better, faster, and more numerous transistors every 18 months. More recently, the end of Dennard scaling has limited the benefits of transistor scaling, resulting in greater concerns about power density and an increased focus on energy efficient architectural mechanisms [SAW$^+$10, FM11]. As benefits of transistor scaling diminish and Moore's law begins to slow, an emphasis is being placed on both *hardware specialization* and *vertically integrated hardware design* as alternative approaches to achieve high-performance and energy-efficient computation for emerging applications.

1.1   Challenges of Modern Computer Architecture Research

These new technology trends have created numerous challenges for academic computer architects researching the design of next-generation computational hardware. These challenges include:

1. **Accurate power and energy modeling:** Credible computer architecture research must provide accurate evaluations of power, energy, and area, which are now primary design constraints. Unfortunately, evaluation of these design characteristics is difficult using traditional computer architecture simulation frameworks.

2. **Rapid design, construction, and evaluation of systems-on-chip:** Modern systems-on-chip (SOCs) have grown increasingly complex, often containing multiple asymmetric processors, specialized accelerator logic, and on-chip networks. Productive design and evaluation tools are needed to rapidly explore the heterogeneous design spaces presented by SOCs.

3. **Effective methodologies for vertically integrated design:** Opportunities for significant improvements in computational efficiency and performance exist in optimizations that reach across the hardware and software layers of the computing stack. Computer architects need productive hardware/software co-design tools and techniques that enable incremental refinement of specialized components from software specification to hardware implementation.

1

Computing Stack
Application
Algorithm
Programming Language
Operating System
Instruction Set Architecture
Microarchitecture
Register-Transfer Level
Gate Level
Circuits
Transistors

Industry Product Capabilities

Academic Project Capabilities
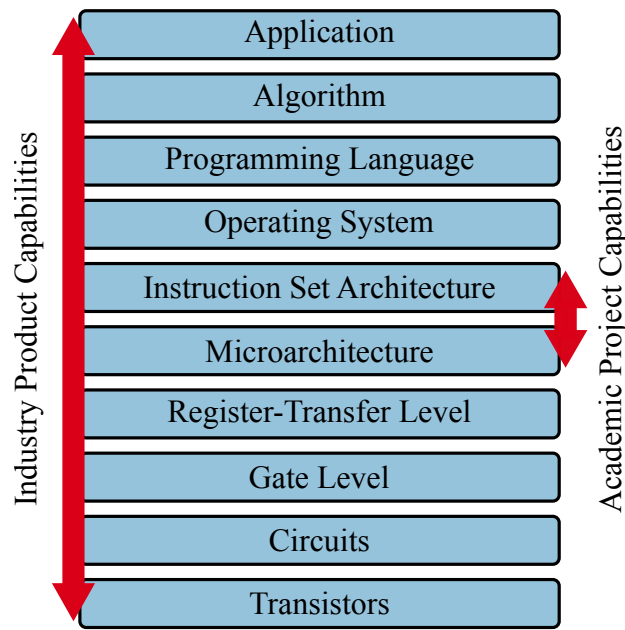
Figure 1.1: The Computing Stack – A simplified view of the computing stack is shown to the left. The *instruction set architecture* layer acts as the interface between software (above) and hardware (below). Each layer exposes abstractions that simplify system design to the layers above, however, productivity advantages afforded by these abstations come at the cost of reduced performance and efficiency. Vertically integrated design performs optimizations across layers and is becoming increasingly important as a means to improve system performance. Academic research groups, traditionally limited to exploring one or two layers of the stack due to limited resources, face considerable challenges performing vertically integrated hardware research going forward.

Industry has long dealt with these challenges through the use of significant engineering resources, particularly with regards to manpower. As indicated in Figure 1.1, the allocation of numerous, specialized engineers at each layer of the computing stack has allowed companies such as IBM and Apple to capitalize on the considerable benefits of vertically integrated design and hardware specialization. In some cases, these solutions span the entire technology stack, including user-interfaces, operating systems, and the construction of application-specific integrated circuits (ASICs). However, vertically integrated optimizations are much less commonly explored by academic research groups due to their greater resource limitations. This trend is likely to continue without considerable innovation and drastic improvements in the productivity of tools and methodologies for vertically integrated design.

1.2   Enabling Academic Exploration of Vertical Integration

In an attempt to address some of these limitations, this thesis demonstrates a novel approach to constructing productive hardware design methodologies that combines embedded domain-specific languages with just-in-time optimization. Embedded domain-specific languages (EDSLs) enable improved designer productivity by presenting concise abstractions tailored to suit the particular needs of domain-specific experts. Just-in-time optimizers convert these high-level EDSL

descriptions into high-performance, executable implementations at run-time through the use of kernel-specific code generators. Prior work on *selective embedded just-in-time specialization* (SEJITS) introduced the idea of combining EDSLs with kernel- and platform-specific JIT specializers for specialty computations such as stencils, and argued that such an approach could bridge the performance-productivity gap between productivity-level and efficiency-level languages [CKL+09]. This work demonstrates how the ideas presented by SEJITS can be extended to create productive, vertically integrated hardware design methodologies via the construction of EDSLs for hardware modeling along with just-in-time optimization techniques to accelerate hardware simulation.

1.3   Thesis Proposal and Overview

This thesis presents two prototype software frameworks, PyMTL and Pydgin, that aim to address the numerous productivity challenges associated with researching increasingly complex hardware architectures. The design philosophy behind PyMTL and Pydgin is inspired by many great ideas presented in prior work, as well as my own proposed computer architecture research methodology I call *modeling towards layout* (MTL). These frameworks leverage a novel design approach that combines Python-based, embedded domain-specific languages (EDSLs) for hardware modeling with just-in-time optimization techniques in order to improve designer productivity and achieve good simulation performance.

Chapter 2 provides a background summary of hardware modeling abstractions used in hardware design and computer architecture research. It discusses existing taxonomies for classifying hardware models based on these abstractions, discusses limitations of these taxonomies, and proposes a new methodology that more accurately represents the tradeoffs of interest to computer architecture researchers. Hardware design methodologies based on these various modeling tradeoffs are introduced, as is the *computer architure research methodology gap* and my proposal for the vertically integrated *modeling towards layout* research methodology.

Chapter 3 discusses the PyMTL framework, a Python-based framework for enabling the *modeling towards layout* evaluation methodology for academic computer architecture research. This chapter discusses the software architecture of PyMTL's design including a description of the PyMTL EDSL. Performance limitations of using a Python-based simulation framework are char-

acterized, and SimJIT, a proof-of-concept, just-in-time (JIT) specializer is introduced as a means to address these performance limitations.

Chapter 4 introduces Pydgin, a framework for constructing fast, dynamic binary translation (DBT) enabled instruction set simulators (ISSs) from simple, Python-based architectural descriptions. The Pydgin architectural description language (ADL) is described, as well as how this embedded-ADL is used by the RPython translation toolchain to automatically generate a high-performance executable interpreter with embedded JIT-compiler. Annotations for JIT-optimization are described, and evaluation of ISSs for three ISAs are provided.

Chapter 5 describes preliminary work on further extensions to the PyMTL framework. An experimental Python-based tool for performing high-level synthesis (HLS) on PyMTL models is discussed. Another tool for creating layout generators and enabling physical design from within PyMTL is also introduced.

Chapter 6 concludes the thesis by summarizing its contributions and discussing promising directions for future work.

1.4 Collaboration, Previous Publications, and Funding

The work done in this thesis was greatly improved thanks to contributions, both small and large, by colleagues at Cornell. Sean Clark and Matheus Ogleari helped with initial publication submissions of PyMTL v0 through their development of C++ and Verilog mesh network models. Edgar Munoz and Gary Zibrat built valuable models using PyMTL v1. Gary additionally was a great help in running last-minute simulations for [LZB14]. Kai Wang helped build the assembly test collection used to debug the Pydgin ARMv5 instruction set simulator and also explored the construction of an FPGA co-simulation tool for PyMTL. Yunsup Lee sparked the impromptu "code sprint" that resulted in the creation of the Pydgin RISC-V instruction set simulator and provided the assembly tests that enabled its construction in under two weeks. Carl Friedrich Bolz and Maciej Fijałkowski provided assistance in performance tuning Pydgin and gave valuable feedback on drafts of [LIB15].

Especially valuable were contributions made by my labmates Shreesha Srinath and Berkin Ilbeyi, and my research advisor Christopher Batten. Shreesha and Berkin were the first real users of PyMTL, writing numerous models in the PyMTL framework and using PyMTL for architectural

exploration in [SIT$^+$14]. Berkin was a fantastic co-lead of the Pydgin framework, taking charge of JIT optimizations and also performing the thankless job of hacking cross-compilers, building SPEC benchmarks, running simulations, and collecting performance results. Shreesha was integral to the development of a prototype PyMTL high-level synthesis (HLS) tool, providing expertise on Xilinx Vivado HLS, a collection of example models, and assistance in debugging.

Christopher Batten was both a tenacious critic and fantastic advocate for PyMTL and Pydgin, providing guidance on nearly all aspects of the design of both frameworks. Particularly valuable were Christopher's research insights and numerous coding "experiments", which led to crucial ideas such as the use of greenlets to create pausable adapters for PyMTL functional-level models.

CHAPTER 2
HARDWARE MODELING FOR COMPUTER ARCHITECTURE RESEARCH

The research, development, and implementation of modern computational hardware involves complex design processes which leverage extensive software toolflows. These design processes, or *design methodologies*, typically involve several stages of manual and/or automated transformation in order to prepare a hardware model or implementation for final fabrication as an application-specific integrated circuit (ASIC) or system-on-chip (SOC). In the later stages of the design process, the terminology used for hardware modeling is largely agreed upon thanks to the wide usage of very-large scale integration (VLSI) toolflows provided by industrial electronic-design automation (EDA) vendors. However, there is much less agreement on terminology to categorize models at higher levels of abstraction; these models are frequently used in computer architecture research where a wider variety of techniques and tools are used.

This chapter aims to provide background, motivation, and a consistent lexicon for the various aspects of hardware modeling and simulation related to this thesis. While considerable existing terminology exists in the area of hardware modeling and simulation, many terms are vague, confusing, used inconsistently to mean different things, or generally insufficient. The following sections describe how many of these terms are used in the context of prior work, and, where appropriate, present alternatives that will be used throughout the thesis to describe my own work.

2.1   Hardware Modeling Abstractions

Hardware modeling abstractions are used to simplify the creation of hardware models. They enable designers to trade-off implementation time, simulation speed, and model detail to minimize time-to-solution for a given task. Based on these abstractions, *hardware modeling taxonomies* have been developed in order to classify the various types of hardware models used during the process of design-space exploration and logic implementation. These taxonomies allow stakeholders in the design process to communicate precisely about what abstractions are utilized by a particular model and implicitly convey what types of trade-offs the model makes. In addition, taxonomies enable discussions about methodologies in terms of the specific model transformations performed by manual and automated design processes. Several taxonomies have been proposed in prior literature to categorize the abstractions used in hardware design, a few of which are described below.

6

(a) Y-Chart Diagram
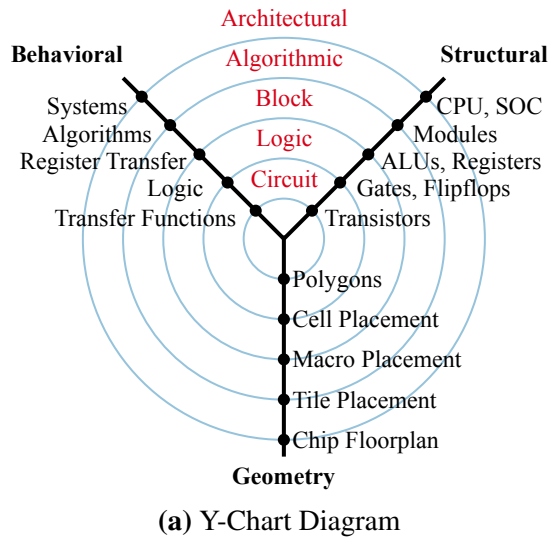
Abstraction Level	Structural Domain	Behavioral Domain	Physical Domain
Functional		algorithm; instruction set	
Architectural	processors, memories, networks	functional with cycle-level timing	macro floorplan
Register-Transfer	dpath/ctrl split; regs, SRAMs, functional units	sequential & combinational concurrent blocks	micro floorplan
Gate	logic gates; flip-flops	boolean equations; truth tables	cell tiling

(b) Table Representation of an Alternative Y-Chart

Figure 2.1: Y-Chart Representations – Originally introduced in [GK83], the Y-chart can be used to classify models based on their characteristics in the structural, behavioral, and geometric (i.e., physical) domains. The traditional Y-chart diagram shown in 2.1a is useful for visually demonstrating design processes that gradually transform models from abstract to detailed representations. An alternative view of the Y-chart from a computer architecture perspective is shown in table 2.1b; red boxes indicate how commonly used hardware design toolflows map to the Y-chart axes. Note that these boxes do not map well to the design flows often described in digital design texts. In practice, different toolflows exists for high-level computer architecture modeling (top-left box), low-level logic design (bottom-left box), and physical chip design (far right box).

2.1.1 The Y-Chart

One commonly referenced taxonomy for hardware modeling is the Y-chart, shown in Figure 2.1a. Complex hardware designs generally leverage hierarchy and abstraction to simplify the design process, and the Y-chart aims to categorize a given model or component by the abstraction level used across three distinct axes or domains. The three domains illustrate three views of a digital system: the *structural* domain characterizes how a system is assembled from interconnected subsystems; the *behavioral* domain characterizes the temporal and functional behavior of a system; and the *geometric* domain characterizes the physical layout of a system. In [GK83] the Y-chart was proposed not only as a way to categorize various designs, but also as a way to describe design methodologies using arrows to specify transformations between domains and abstraction levels. Design methodologies illustrated using the Y-chart typically consist of a series of arrows that iteratively work their way from the more abstract representations located on the outer rings down to the detailed representations on the inner rings.

(a) Design Cube Axes

(b) Design Classifications
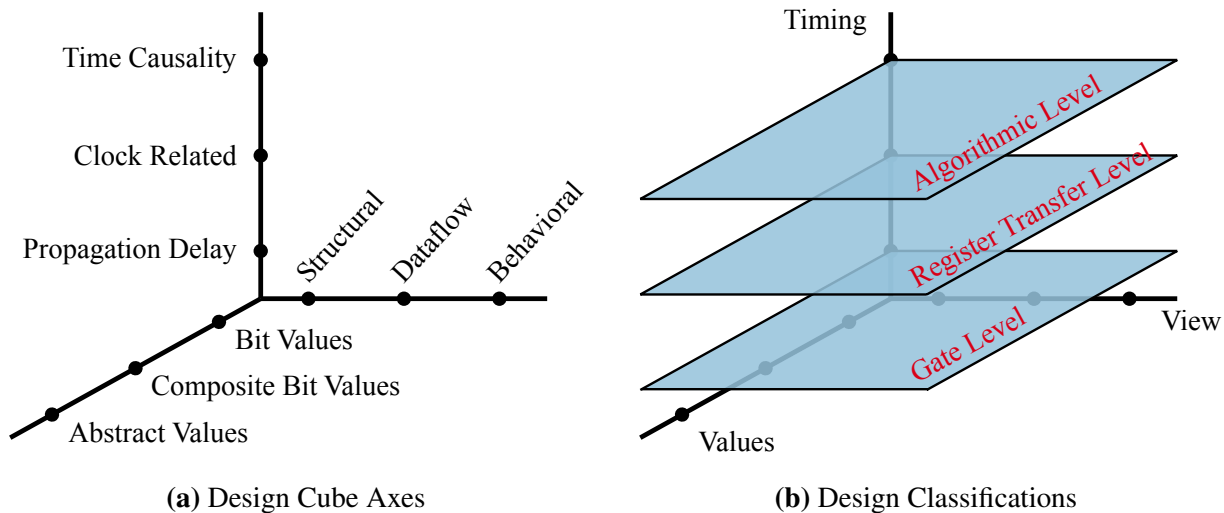
Figure 2.2: Eckert Design Cube – The above diagrams demonstrate the axes, abstractions, and design classifications of the Design Cube as presented in [EH92]. Specific VHDL model instances can be plotted as points within the design cube space based on their timing, value, and design view abstractions (2.2a); Eckert argued it was often more useful to group these model instances into more general classifications based purely on their timing characteristics (2.2b).

While a useful artifact for thinking about the organization of large hardware projects, the Y-chart does not map particularly well to the methodologies and toolflows used by most computer architects and digital designers. For example, consider the alternative mapping of the Y-chart to an architecture-centric view in Table 2.1b. A typical methodology for hardware design leverages three very different software frameworks for: (1) high-level functional/architectural modeling in the structural and behavioral domains (e.g., SystemC, C/C++); (2) low-level RTL/gate-level modeling in the structural and behavioral domains (e.g., SystemVerilog, VHDL); and (3) modeling in the geometric domain (e.g., TCL floorplanning scripts).

2.1.2 The Ecker Design Cube

One primary criticism of the Y-chart taxonomy is the fact that it is much more suitable for describing a process or path from architectural- to circuit-level implementation than it is for quantifying the state of a specific model. Another significant criticism is the fact that the Y-chart does not map particularly well to hardware modeling languages like Verilog and VHDL which describe behavioral and structural aspects of design, but not geometry. To address these deficiencies, Eckert and Hofmeister presented the Design Cube as an alternative taxonomy for VHDL models [EH92].

Value ──────────●──────────────────●──────────────▶
        True   Partially True

Format ────●───────────────●──────────────────●───▶
        Bit   Composite Bit   Abstract

Timing ───●────────●────────●────────●────────●────●──▶
     Gate Delay   Clock   Wallclock   Generic   Event/   Causality
              Related   Message

State ────●──────────────●──────────────●──────────────●──▶
     Complete Internal   Program Visible   Partial   State
          State   State   State   Not Modeled
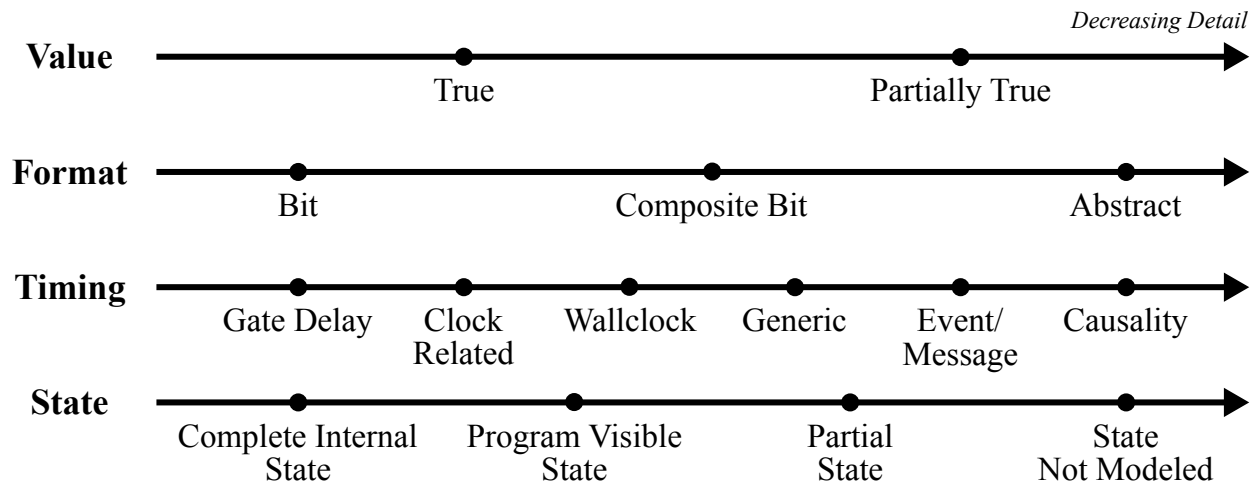
Figure 2.3: Madisetti Taxonomy Axes – Four axes of classification were proposed by Madisetti in [Mad95], two of which (value and format) categorize the accuracy of datatypes used within a model. Although not explicitly indicated in the diagram above, different classifications could potentially be assigned to the kernel and the interface of a model depending on how much detail was tracked internally versus exposed externally.

The Design Cube, shown in Figure 2.2, specifies three axes: *design view*, *timing*, and *value*. The design view axis specifies the modeling style used by the VHDL model, either behavioral, dataflow, or structural. The behavioral and structural design views map directly to the behavioral and structural domains of the Y-chart, while dataflow is described as a bridge between these two views. The timing and value axes describe the abstraction level of timing information and data values represented by the model, respectively. Using these three axes, models can be classified as discrete points within the design cube space, and design processes can be described as edges or transitions between these points. [EH92] additionally proposed a "design level" classification for models based on the timing axis. A diagram of this classification can be seen in Figure 2.2b.

2.1.3 Madisetti Taxonomy

A taxonomy by Madisetti was proposed as a means to classify the fidelity of VHDL models used for virtual prototyping [Mad95]. The four axes of classification in Madisetti's taxonomy, shown in Figure 2.3, are meant to categorize not just hardware but also module interaction with co-designed software components. Both the *value* and *format* axes are used to described the fidelity of datatypes used within a model: the value axis describes signals as either *true* or *partially true* depending on their numerical accuracy, while the format axis describes the representation

Temporal Resolution

Higher Resolution   *Lower Resolution*

⟵●————————●————————●————————●————————●————————●————————●⟶

| Gate Propagation | Clock Accurate | Cycle Approximate | Instruction Cycle | Token Cycle | System Event | Partial Order |

Data Resolution

⟵●————————●————————●————————●————————●⟶

| Bit | Format | Value | Property | Token |

Functional Resolution

⟵●————————————————●————————————————●⟶

| Digitial Logic | Algorithmic | Mathematical |

Structural Resolution

⟵●————————————————●————————————————●⟶

| Structural | Block Diagram | Single Black Box |

Figure 2.4: RTWG/VSIA Taxonomy Axes – The RTWG/VSIA taxonomy presented in [BMGA05] takes influence from, and expands upon, many of the ideas proposed by the Y-chart, Design Cube, and Madisetti taxonomies. The four axes provide separate classifications of the internal state and external interface of a model. A fifth axes, not shown above, was also proposed to describe the *software programmibility* of a model, i.e., how it appears to target software.

abstraction used by signals (*bit*, *composite bit*, or *abstract*). The *timing* axis classifies the detail of timing information provided by a model. The *state* axis describes the amount of internal state information tracked and exposed to users of the model.

An interesting aspect of Madisetti's proposed taxonomy is that it provides two distinct classifications for a given model: one for the kernel (datapath, controllers, storage) and another for the interface (ports). One benefit of this approach is that the interoperability of two models can be easily determined by ensuring the timing and format axes of their interface classifications intersect.

2.1.4 RTWG/VSIA Taxonomy

The RTWG/VSIA taxonomy, described in great detail by [BMGA05], evolved from the combined efforts of the RASSP Terminology Working Group (RTWG) and the Virtual Socket Interface Aliance (VSIA). Initial work on this taxonomy came from the U.S. Department of Defense funded Rapid Prototyping of Application Specific Signal Processors (RASSP) program. It was later refined by the industry-supported VSIA in hopes of clarifying the modeling terminology used within

Taxonomy	Axes					
Y-Chart			Structural	Functional	Geometric	
Design Cube	Timing	Value	View			
Madisetti	Timing	Format				Value	State
RTWG/VSIA	Temporal Resolution	Data Value	Structural Resolution	Functional Resolution			Internal/ External

Table 2.1: Comparison of Taxonomy Axes – Reproduced from [BMGA05], the above table compares the classification axes used by each taxonomy. Note that Madisetti specifies *value* to have a meaning that is different from the Design Cube and RTWG/VSIA taxonomies.

the IC design community. Figure 2.4 shows the four axes that the RTWG/VSIA taxonomy uses to classify models: *temporal resolution*, *data resolution*, *functional resolution*, and *structural resolution*. Like the Madisetti taxonomy, the RTWG/VSIA taxonomy is intended to apply the four axes independently to the internal and external views of a model, effectively grading a model on eight attributes. An additional axis called the *software programming* axis, not shown in Figure 2.4, is also proposed by the RTWG/VSIA taxonomy in order to describe the interfacing of hardware models with co-designed software components.

A comparison of the concepts used by the RTWG/VSIA taxonomy with the taxonomies previously discussed can be seen in Table 2.1. Note that the structural resolution and functional resolution axes mirror the structural and functional axes of the Y-chart, while the temporal resolution and data resolution axes mirror the timing and value axes used in the Ecker Design Cube.

Also defined in [BMGA05] is precise terminology for a number of model classes widely used by the hardware design community, along with their categorization within this RTWG/VSIA taxonomy. A few of these model classifications are summarized below:

- **Functional Model** – describes the function of a component without specifying any timing behavior or any specific implementation details.

- **Behavioral Model** – describes the function and timing of a component, but does not describe a specific implementation. Behavioral models can come in a range of abstraction levels; for example, *abstract-behavioral* models emulate cycle-approximate timing behavior and expose inexact interfaces, while *detailed-behavioral* models aim to reproduce clock-accurate timing behavior and expose an exact specification of hardware interfaces.

- **Instruction-Set Architecture Model** – describes the function of a processor instruction set architecture by updating architecturally visible state on an instruction-level granularity. In the RTWG/VSIA taxonomy, a processor model without ports is classified as an ISA model, whereas a processor model with ports is classified as a behavioral model.

- **Register-Transfer-Level Model** – describes a component in terms of combinational logic, registers, and possibly state-machines. Primarily used for developing and verifying the logic of an IC component, an RTL model acts as unambiguous documentation for a particular design solution.

- **Logic-Level Model** – describes the function and timing of a component in terms of boolean logic functions and simple state elements, but does not describe details of the exact logic gates needed to implement the functions.

- **Cell-Level Model** – describes the function and timing of a component in terms of boolean logic gates, as well as the structure of the component via the interconnections between those gates.

- **Switch-Level Model** – describes the organization of transistors implementing the behavior and timing of a component; the transistors are modeled as voltage-controlled on-off switches.

- **Token-Based Performance Model** – describes performance of a system's architecture in terms of response time, throughput, or utilization by modeling only control information, not data values.

- **Mixed-Level Model** – is a composition of several models at different abstraction levels.

2.1.5 An Alternative Taxonomy for Computer Architects

A primary drawback of the previous taxonomies is that they do not clearly convey the attributes computer architects care most about when building and discussing models. Using the Y-chart as an example, the *structural* and *behavioral* domains dissociate the functionality aspects of a model since the same functional behavior can be achieved from either a monolithic or hierarchical design (in an attempt to remedy this, the Design Cube combined these two attributes into a single axis). In the RTWG/VSIA taxonomy, *data resolution* is its own axis although it has overlap with both the *functional resolution* axis when computation is approximate and the *resource resolution* axis

12

Behavioral Accuracy

Lower Resolution   *Higher Resolution*

None	Approximate	Precise

Timing Accuracy

None	Instruction/ Event	Cycle Approximate	Cycle Precise	Gate Propagation

Resource Accuracy

None	Monolithic	Unit-Level	Gate-Level	Transistor-Level
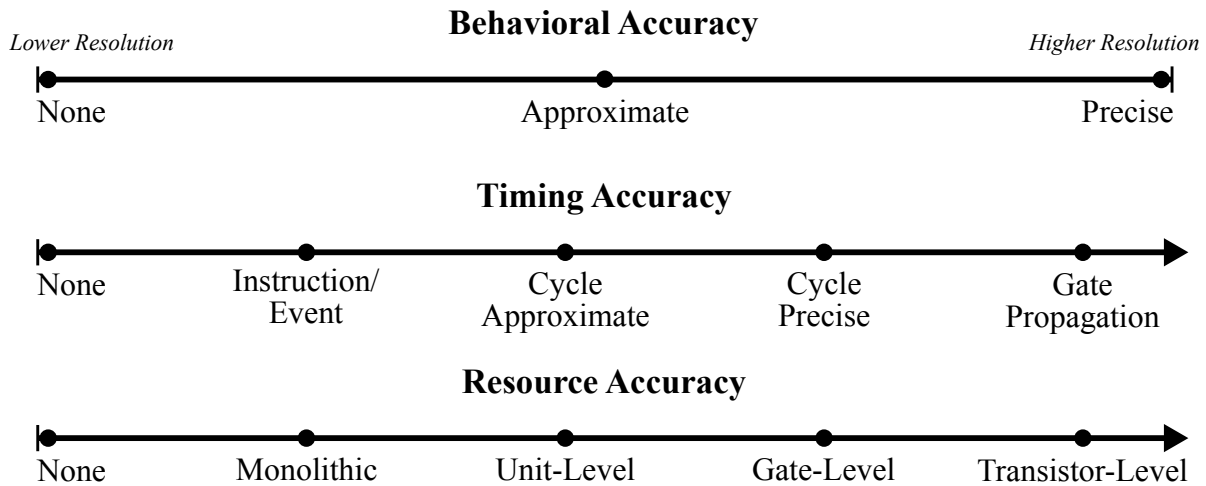
Figure 2.5: A Taxonomy for Computer Architecture Models – The proposed taxonomy above characterizes hardware models based on how precisely they model the behavior, timing, and resources of target hardware. In this context, behavior refers to the *functional* behavior of a model: how input values map to output values. These axes map well to three important model classes used by computer architects: functional-level (FL), cycle-level (CL), and register-transfer-level (RTL).

when computation is bit-accurate. Similarly, the *structural* and *geometric* domains of the Y-chart both hint at how accurately a model represents physical hardware resources, however, physical geometry generally plays little role in the models produced by computer architects.

In addition, the model classifications suggested by some of the previous taxonomies do not map particularly well to the model classes most-frequently used by computer architects. For example, the classifications suggested by the Ecker Design Cube in Figure 2.2b do not include higher-level, timing agnostic models. These classifications are based only on the timing axis of the Design Cube and do not consider the relevance of other attributes fundamental to hardware modeling.

To address these issues, an alternative taxonomy is proposed which classifies hardware models based on three attributes of primary importance to computer architects: *behavioral accuracy*, *timing accuracy*, and *resource accuracy*. The abstractions associated with each of these axes are shown in Figure 2.5 and described in detail below:

- **Behavioral Accuracy** – describes how correctly a model reproduces the functional behavior of a component, i.e., the accuracy of the generated outputs given a set of inputs. In most cases computer architects want the functional behavior of a model to precisely match the target hardware. Alternatively a model designer may want behavior that only approximates the target hardware (e.g., floating-point reference models used to track the error of fixed-point

target hardware), or may not care about the functional behavior at all (e.g., analytical models that generate timing or power estimates).

- **Timing Accuracy** – describes how precisely a model recreates the timing behavior of a component, i.e., the delay between when inputs are provided and the output becomes available. Computer architects generally strive to create models that are *cycle precise* to the target hardware, but in practice their models are typically more correctly described as *cycle approximate*. Models that only track timing on an event-level basis are also quite common (e.g., instruction set simulators). Models with finer timing granularity than cycle level are sometimes desirable (e.g., gate-level simulation), but such detail is rarely necessary for most computer architecture experiments.

- **Resource Accuracy** – describes to what degree a model parallels the physical resources of a component. These physical resources include both the granularity of component boundaries as well as the structure of interface connections. Accurate representation of physical resources generally make it easier to correctly replicate the timing behavior of a component, particularly when resources are shared and can only service a limited number of requests in a given cycle. *Structural-concurrent* modeling frameworks and hardware-description languages (HDLs) make component and interface resources an explicit first-class citizen, greatly simplifying the task of accurately modeling the physical structure of a design; functional and object-oriented modeling frameworks have no such notion and require extra diligence by the designer to avoid unrealistic resource sharing and timing behavior [VVP$^+$02, VVP$^+$06].

Note that the abstraction levels for all three axes begin with *None* since it is sometimes desirable for a model to convey no information about a particular axis.

Three model classes widely used in computer architecture research map particularly well to the axes described above: *functional-level (FL)* models imitate just the behavior of target hardware, *cycle-level (CL)* models imitate both the behavior and timing, and *register-transfer-level (RTL)* models imitate the behavior, timing, and resources. Figure 2.6 shows how the FL, CL, and RTL classes map to the behavioral accuracy, timing accuracy, and resource accuracy axes. Note that for each model class there is often a range of accuracies at which it may model each attribute. The common use cases and implementation strategies for each of these models are described in greater detail below.

- **Functional-Level (FL)** – models implement the *functional behavior* but not the timing constraints of a target. FL models are useful for exploring algorithms, performing fast emulation of hardware targets, and creating golden models for validation of CL and RTL models. The FL methodology usually has a data structure and algorithm-centric view, leveraging productivity-level languages such as MATLAB or Python to enable rapid implementation and verification. FL models often make use of open-source algorithmic packages or toolboxes to aid construction of golden models where correctness is of primary concern. Performance-oriented FL models may use efficiency-level languages such as C or C++ when simulation time is the priority (e.g., instruction set simulators).

- **Cycle-Level (CL)** – models capture the *behavior* and *cycle-approximate timing* of a hardware target. CL models attempt to strike a balance between accuracy, performance, and flexibility while exploring the timing behavior of hypothetical hardware organizations. The CL methodology places an emphasis on simulation speed and flexibility, leveraging high-performance efficiency-level languages like C++. Encapsulation and reuse is typically achieved through classic object-oriented software engineering paradigms, while timing is most often modeled using the notion of ticks or events. Established computer architecture simulation frameworks (e.g., ESESC [AR13], gem5 [BBB⁺11]) are frequently used to increase productivity as they typically provide libraries, simulation kernels, and parameterizable baseline models that allow for rapid design-space exploration.

- **Register-Transfer-Level (RTL)** – models are *behavior-accurate*, *cycle-accurate*, and *resource-accurate* representations of hardware. RTL models are built for the purpose of verification and synthesis of specific hardware implementations. The RTL methodology uses dedicated hardware description languages (HDLs) such as SystemVerilog and VHDL to create bit-accurate, synthesizable hardware specifications. Language primitives provided by HDLs are designed specifically for describing hardware: encapsulation is provided using port-based interfaces, composition is performed via structural connectivity, and logic is described using combinational and synchronous concurrent blocks. These HDL specifications are passed to simulators for evaluation/verification and EDA toolflows for collection of area, energy, timing estimates and construction of physical FPGA/ASIC prototypes. Originally intended for the design and verification of individual hardware instances, traditional HDLs are not well suited for extensive design-space exploration [SAW⁺10, SWD⁺12, BVR⁺12].

15
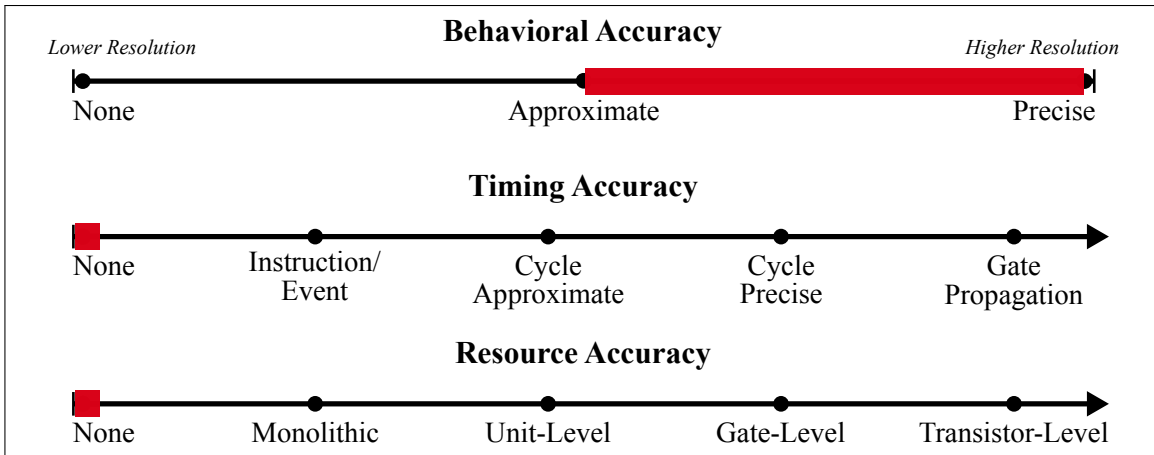
(a) Functional-Level (FL) Model
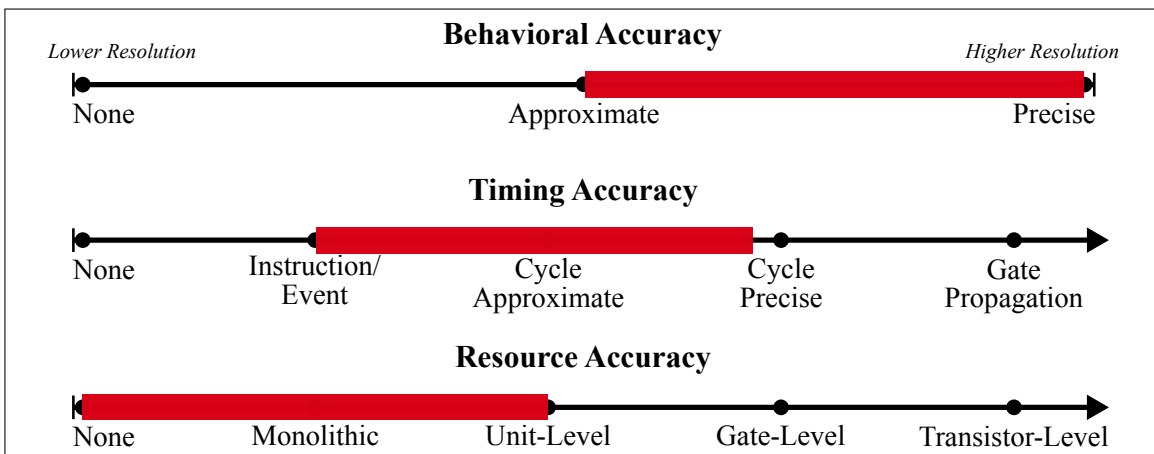
(b) Cycle-Level (CL) Model
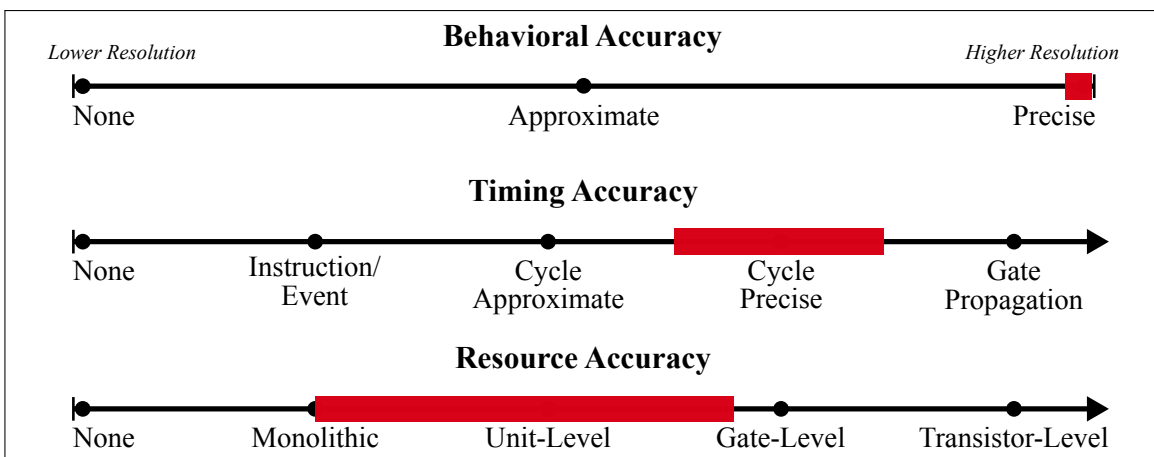
(c) Register-Transfer-Level (RTL) Model

Figure 2.6: Model Classifications – The FL, CL, and RTL model classes each model a component's behavior, timing, and resources to different degrees of accuracy.

16

2.1.6   Practical Limitations of Taxonomies

Although the taxonomy proposed in the previous section maps much more directly to the models and research methodologies used by most computer architects, it does not address many of the practical, software-engineering issues related to model implementations. Two models may have identical classifications with respect to each of their axes, however, they may be incompatible due to the use of different implementation approaches (for example, the use of port-based versus method-based communication interfaces). This is particularly problematic within the context of a design process that leverages multiple design languages and simulation tools. As previously indicated for the Y-chart in Figure 2.1b and the functional-, cycle-, and register-transfer level models in Section 2.1.5, transformations along and/or across axes boundaries, or between model classes, often require the use of multiple distinct toolflows. Research exploring vertically integrated architectural optimizations encounter these boundaries frequently, and as will be discussed in Section 2.2, this context switching between various languages, design patterns, and tools can be a significant hindrance to designer productivity. A few of the software-engineering challenges facing computer architects wishing to build models for co-simulation are discussed below.

Model Implementation Language   Ideally, two interfacing models would use an identical modeling language, but if models are at different levels of abstraction, this may not be the case. An ordering of possible interfacing approaches from easiest to most difficult includes: *identical language*, *identical runtime*, *foreign-function interface*, and *sockets/files*.

Model Interface Style   Models written in different languages and frameworks may use different mechanisms for their communication interfaces. For example, components written in hardware description languages expose ports in order to communicate inputs and outputs, while components in object-oriented languages usually expose methods instead. Some different styles of model interfacing include *ports*, *methods*, and *functions*.

Model Composition Style   The interface style also strongly influences the how components are composed in the model hierarchy. Models with port-based interfaces use *structural composition*: input values are received from input ports that are physically connected with the output ports of another component; output values are returned using output ports which are again physically con-

nected to the input ports of another component. These structural connections prevent a component from being reused by multiple producers unless (a) multiple unique instances are created and individually connected to each producer or (b) explicit arbitration units are used to interface the multiple producers with the component. In contrast, models with method- or function-based interfaces use *functional composition*: input values are received via arguments from a single caller, and output values are generated as a return value to the same caller. This call interface can be reused by multiple producers without the need for arbitration logic, often unintentionally. Structural composition inherently limits models to having at most one parent, whereas functional composition allows models to have multiple parents and global signals that violate encapsulation.

Model Logic Semantics   Different modeling languages also use different execution semantics for their logic blocks. Hardware description languages provide blocks with concurrent execution semantics to better match the behavior of real hardware. These concurrent blocks can execute either synchronously or combinationally. Most popular general-purpose languages have sequential execution semantics and function calls in these languages are non-blocking, although it is also possible to leverage asynchronous libraries to provide blocking semantics. Logic execution semantics for hardware models are generally one of the following: *concurrent synchronous*, *concurrent combinational*, *sequential non-blocking*, or *sequential blocking*.

Model Data Types   Models constructed using the same programming language and communication interface may still be incompatible because they exchange values using different data types. Data types typically must share both the same *structure* and *encoding* in order to be compatible. The structure of a data type describes how it encapsulates and provides access to data; a data type structure may simply be a *raw value* (e.g., `int`, `float`), it may have *fields* (e.g., `struct`), or it may have *methods* (e.g., `class`). The encoding of a data type describes how the value or values it encapsulates are represented, which could potentially be *strings/tokens*, *numeric*, or *bit-accurate*.

2.2   Hardware Modeling Methodologies

Current computer architecture research involves using a variety of modeling languages, modeling design patterns, and modeling tools depending on the level of abstraction a designer is working

18

	FL	**CL**	**RTL**
Modeling Languages	Productivity Level (PLL)	Efficiency Level (ELL)	Hardware Description (HDL)
	MATLAB/R/Python	*C/C++*	*Verilog/VHDL*
Modeling Patterns	Functional: Data Structures, Algorithms	Object-Oriented: Classes, Methods, Ticks and/or Events	Concurrent-Structural: Combinational Logic, Clocked Logic, Port Interfaces
Modeling Tools	Third-party Algorithm Packages and Toolboxes	Computer Architecture Simulation Frameworks	Simulator Generators, Synthesis Tools, Verification Tools

Table 2.2: Modeling Methodologies – Functional-level (FL), cycle-level (CL), and register-transfer-level (RTL) models used by computer architects each have their own methodologies with different languages, design patterns, and tools. These distinct methodologies make it challenging to create a unified modeling environment for vertically integrated architectural exploration.

at. These languages, design patterns, and tools can be used to describe a *modeling methodology* for a particular abstraction level or model class. A summary of the modeling methodologies for the functional-level (FL), cycle-level (CL), and register-transfer-level (RTL) model classes introduced in Section 2.1.5 is shown in Table 2.2.

Learning the languages and tools of each methodology requires a significant amount of intellectual overhead, leading many computer architects to specialize for the sake of productivity. An unfortunate side-effect of this overhead-induced specialization has been a split in the computer architecture community into camps centered around the research methodologies and toolflows they use. Two particularly pronounced camps are those centered around the cycle-level (CL) and register-transfer-level (RTL) research methodologies. These methodologies, as well as the functional-level (FL) methodology, are described in the subsections below.

As vertically integrated design becomes of greater importance for achieving performance and efficiency goals, the challenges associated with context switching between the FL, CL, and RTL methodologies will become increasingly prominent. A few possible approaches for constructing vertically integrated research methodologies that address some of these challenges are discussed below. One particularly promising methodology which has been adopted by the PyMTL framework, called *modeling towards layout* (MTL), is also introduced.

2.2.1 Functional-Level (FL) Methodology

Computer architecture and VLSI researchers widely use the functional-level (FL) methodology to both create "golden" reference models for validation and to perform exploratory algorithmic experimentation. The FL methodology frequently takes advantage of *productivity-level languages* (PLLs) such as Matlab, R, and Python to enable rapid model construction and experimentation. These languages have higher-level language constructs and provide access to a wide array of algorithmic toolboxes and packages; these packages are either included as part of the language's standard library or made available by third-parties. One drawback of PLLs is that they generally exhibit much slower simulation performance than *efficiency-level languages* (ELLs) such as C and C++. While the rising popularity of PLLs has resulted in the active development of JIT compilers that significantly improve the execution performance of these languages [CBHV10, RHWS12, BCFR09], they may not be suitable for all types of FL models. For example, instruction set simulators, which are functional models that model the instruction-level execution behavior of a processor architecture, must be extremely fast to execute large binaries. Instruction-set simulators are nearly always implemented in C or C++ and often have complex implementations that utilize advanced performance optimizations like dynamic binary translation.

The FL methodology is becoming more important as algorithmic optimizations are mapped into hardware to create specialized units, requiring extensive design-space exploration at the algorithm level. However, FL models' lack of timing and resource information require their use in tandem with CL or RTL models in order to perform computer architecture studies that propose microarchitectural enhancements.

2.2.2 Cycle-Level (CL) Methodology

Modern computer architecture research has increasingly relied on the *cycle-level* (CL) methodology as the primary tool for evaluating novel architectural mechanisms. A CL methodology is characterized as the use of a *cycle-approximate simulator* (CAS), generally in the form of a simulation framework implemented in a general-purpose language such as C or C++ (e.g., gem5 [BBB$^+$11], SimpleScalar [ALE02], ESESC [AR13]), configured and/or modified to model a particular system architecture and any enhancements. Models built using a CL methodology are capable of generating fairly accurate estimates of system performance (in terms of cycles ex-

ecuted) provided that the simulated models properly implement a sufficient level of architectural detail [BGOS12]. Additional tools like McPAT [LAS$^+$13], CACTI [MBJ09], Wattch [BTM00], ORION [WPM02, KLPS09, KLPS11], DSENT [SCK$^+$12], and Aladdin [SRWB14] can augment these frameworks to generate preliminary estimates for other key metrics such as area, energy, and timing. The architecture community has generally considered the benefits (e.g., flexibility, simulation speed) of CAS frameworks to outweigh the drawbacks (e.g., accuracy, reproducibility), making them a popular choice among academic research groups.

One considerable limitation of the CL methodology is its limited credibility when generating performance estimates for novel architectures such as specialized accelerators. A CL methodology relies on existing implementations of processors, memories, and networks to act as targets from which models implemented in a CAS framework are validated. However, for targets such as novel accelerators, researchers must model large, unique, and workload-specific hardware blocks that generally have no hardware targets from which to validate their CL models. Several studies have shown that without validation, performance and power estimates from CL models become difficult to trust [GKO$^+$00, GKB09, BGOS12, GPD$^+$14, DBK01, CLSL02]. In addition, specialization research is particularly interested in tradeoffs related to area and energy-efficiency, making it particularly difficult to perform credible specialization research using a purely CL methodology. While tools like Aladdin can help explore the area and power tradeoffs of accelerator architectures, they are not designed for characterizing more programmable specialized units. Due to their high-productivity but limited accuracy, CL methodologies are best suited for performing rapid and extensive design-space exploration to detect performance trends, rather than generating accurate performance measurements for specific design points.

2.2.3 Register-Transfer-Level (RTL) Methodology

Another methodology for computer architecture research relies on the construction of synthesizable *register-transfer-level* (RTL) models using a *hardware description language* (HDL) such as SystemVerilog, Verilog, or VHDL. Paired with a commercial ASIC EDA toolflow, a synthesizable implementation can generate highly accurate estimates of area, energy, cycle time, and cycles executed. Unlike the approximate models of hardware behavior created using a CL methodology, an RTL methodology constructs a true hardware implementation which provides a number of advantages beyond just improved accuracy. An RTL methodology increases credibility with industry,

21

enforces a model/simulator separation which enhances reproducibility, provides interoperability with a range of industry-grade tools, and enables a path to build FPGA or ASIC prototypes. Although widely used by industry and VLSI research groups, the RTL methodology is less popular in computer architecture research due to long development times and slow RTL simulation speeds.

An RTL methodology can be leveraged by architects to create HDL implementations of novel specialized blocks, helping address some of the problems encountered when experimenting with architectures incorporating large amounts of specialization. Unfortunately, the limited productivity of an HDLs makes it difficult to quickly create hardware instances of many different specialized units. In addition, the limited general-purpose programming capabilities of HDLs make it difficult for researchers to mix blocks at various levels of abstraction, for example, composing detailed models of specialized units with cycle-approximate, analytical, or functional implementations of other system components. Research with RTL methodologies generally focus on generating accurate area, energy, and timing evaluations of particular design *instances*, but perform minimal design-space exploration as a result of the limited productivity of HDL design.

However, the recent development of modern, highly-parameterizable hardware description languages, referred to in this thesis as *hardware generation languages* (HGLs), have enabled considerably increased productivity for RTL methodologies. HGLs such as Genesis2 [SWD+12], Chisel [BVR+12], and Bluespec [Nik04, HA03] allow researchers to replace the traditional practice of constructing specific *chip instances*, with an approach focused on construction of *chip generators*. As described by Schacham et al. , chip generators include templated RTL implementations that enable several key productivity advantages, including rapid design-space exploration and codification of designer knowledge [SAW+10]. While HGLs considerably improve designer productivity, thus increasing opportunities for large design-space exploration while still achieving high-fidelity simulation, their productivity remains limited when compared to CL methodologies due to their fundamentally more detailed, and therefore slower, simulation speeds.

2.2.4 The Computer Architecture Research Methodology Gap

The distinct languages, design patterns, and tools utilized by the FL, CL, and RTL methodologies result in a *computer architecture research methodology gap*. This methodology gap introduces intellectual and technical barriers that make it challenging to transition *between* modeling abstractions and even more challenging to create an integrated flow *across* modeling abstractions.

22

Industry is able to bridge this gap by utilizing their considerable resources to build large design teams. For academic research groups with limited resources and manpower bridging this gap can be exceedingly difficult, often resulting in over-emphasis on a single level of abstraction.

Several mechanisms introduced in prior work have shown promise as techniques to help address the various challenges contributing to the methodology gap. These mechanisms are listed and briefly discussed below.

Concurrent-Structural Frameworks   Concurrent-structural frameworks provide hardware-inspired constructs for modeling port-based interfaces, concurrent execution, and structural composition. Vachharajani et al. have shown these constructs address the *mapping problem* inherent to CL models written in sequential, object-oriented languages, greatly improving clarity, accuracy, and component reuse [VVP+02]. HDLs for RTL design generally provide these constructs natively, however, a few general-purpose language, cycle-level simulation frameworks have adopted similar features (e.g., Liberty [VVP+06, VVP+02], Cascade [GTBS13], and SystemC [Pan01]).

Unified Modeling Languages   Unified modeling languages enable specification of multiple modeling abstractions using a single description language. The use of a single specification language greatly reduces cognitive overhead for designers who would otherwise need expertise in multiple design languages. SystemC was proposed as a C++ language for multiple modeling tasks [Pan01] including FL/CL/transaction-level modeling and RTL design (using a synthesizable subset), but has primarily seen wide adoption for virtual system prototyping and high-level synthesis.

Hardware Generation Languages   *Hardware generation languages* (HGLs) are hardware design languages that enable the construction of highly-parameterizable *hardware templates* [SAW+10]. HGLs facilitate design-space exploration at the register-transfer level; some HGLs also improve the productivity of RTL design through the introduction of higher-level design abstractions. Examples of HGLs include Genesis II [SWD+12], Chisel [BVR+12], and Bluespec [Nik04, HA03].

HDL Integration   HDL integration provides mechanisms for native co-simulation of Verilog/VHDL RTL with FL/CL models written in more flexible general-purpose languages. Such integration accelerates RTL verification by supporting fast multilevel simulation of Verilog components with CL models, and enabling embedding of FL/CL golden models within Verilog for test bench validation.

Grossman et al. noted that the unique HDL integration techniques in the Cascade simulator, such as interface binding, greatly assisted hardware validation of the Anton supercomputer [GTBS13].

SEJITS   Selective embedded just-in-time specialization (SEJITS) pairs embedded domain-specific languages (EDSLs) [Hud96] with EDSL-specific JIT compilers to provide runtime generation of optimized, platform-specific implementations from high-level descriptions. SEJITS enables efficiency-level language (ELL) performance from productivity-level language (PLL) code, significantly closing the *performance-productivity gap* for domain specific computations [CKL⁺09]. As an additional benefit, SEJITS greatly simplifies the construction of new domain-specific abstractions and high-performance JIT specializers by embedding specialization machinery within PLLs like Python.

Latency-Insensitive Interfaces   While more of a best-practice than explicit mechanism, consistent use of latency-insensitive interfaces at module boundaries is key to constructing libraries of interoperable FL, CL, and RTL models. Latency-insensitive protocols provide *control abstraction* through module-to-module stall communication, significantly improving component composability, design modularity, and facilitating greater test re-use [VVP⁺06, CMSV01].

Each of these mechanisms can potentially be integrated into toolflows in order to ease the transition between model abstractions. The remaining two sections discuss vertically integrated research methodologies that could potentially benefit from the use of these mechanisms.

2.2.5   Integrated CL/RTL Methodologies

One approach to enabling more vertically integrated research methodologies involves combining the CL and RTL methodologies to enable test reuse and co-simulation of CL and RTL models. Such an integrated CL/RTL methodology would allow the productivity and simulation benefits of cycle-approximate simulation with the credibility of RTL designs. Prior work has shown such integration provides opportunities for improved design-space exploration and RTL verification [GTBS13].

Transitioning between CL and RTL models in such a methodology would still be a manual process, such as when refining a CL model used for design space exploration into a more detailed RTL

implementation. Converting from a model built in a CAS framework into a synthesizable HDL implemenation is currently a non-trivial process, requiring a complete rewrite in a new language and familiarity with a wholly different toolflow. HGLs can greatly help with the productivity of creating these RTL designs, however, they do not address the problem of interfacing CL and RTL models. Three possible approaches for enabling such integration are outlined below.

Integrating HGLs into Widely Adopted CAS Frameworks A relatively straight-forward approach involves translating HGL design instances into industry standard HDLs and then compiling this generated HDL into a widely adopted CAS framework. For example, one could use Chisel to implement a specialized block, generate the corresponding Verilog RTL, use a tool such as Verilator [ver13] to translate the Verilog RTL into a cycle-accurate C++ model, and then link the model into the gem5 simulation framework. One limitation of this approach is that most widely adopted CAS frameworks are not designed with HGL integration in mind, potentially limiting the granularity of integration. While it might be possible to integrate blocks designed with an HGL methodology into the memory system of gem5, it would be more challenging to create new specialized functional units or accelerators that are tightly coupled to the main processor pipeline.

Integrating HGLs into New CAS Frameworks A more radical approach involves developing a new CAS framework from scratch specifically designed to facilitate tight integration with HGLs. Such a CAS framework would likely need to avoid performance optimizations such as split functional/timing models and use some form of concurrent-structural modeling [VVP$^+$02, VVP$^+$06]. One example of such an approach is Cascade, a C++ framework used in the development of the Anton 2 supercomputer. Cascade was specifically designed to enable rapid design-space exploration using a CL methodology while also providing tight integration with Verilog RTL [GTBS13]. Cascade includes support for interfacing binding, enabling composition of Verilog and C++ modules without the need for specialized data-marshalling functions.

Creating a Unified CAS/HGL Framework The most extreme approach involves constructing a completely unified framework that enables construction of both CL and RTL models in a single high-level language. Like Cascade, such a framework would likely have a concurrent-structural modeling approach with port-based interfaces and concurrent execution semantics, as well as provide bit-accurate datatypes, in order to support the design of RTL models. Additionally, such a

framework would need to have a translation mechanisms to convert RTL models described in the framework into an industry-standard HDL to enable compatibility with EDA toolflows.

SystemC was originally envisioned to be such a framework, although it is mostly used in practice for cycle-approximate and even more abstract transaction-level modeling to create virtual system platforms for early software development [sys14]. The PyMTL framework was also designed with this approach in mind, however, it additionally extends its capabilities up the stack to enable construction of FL models as well. The integrated FL/CL/RTL methodology built into PyMTL is described in the next section.

2.2.6 Modeling Towards Layout (MTL) Methodology

While the integrated CL/RTL methodologies described above help provide interoperability between CL and RTL models, thus enabling rapid design space exploration and the collection of credible area/energy/timing results, they do not address the issue of interfacing with FL models. As mentioned previously, FL models are increasingly important in the design of specialized accelerators that map algorithmic optimizations into hardware implementations. Construction of such accelerators benefit considerably from an incremental design strategy that refines a component from high-level algorithm, to cycle-approximate model, to detailed RTL implementation, while also performing abstraction-level appropriate design-space exploration along the way. I call such a modeling methodology *modeling towards layout* (MTL).

The goal of the MTL methodology is to take advantage of the individual strengths of FL, CL, and RTL models, and combine them into a unified, vertically integrated design flow. The hope is that such a methodology will enable computer architects to generate credible analyses of area, energy, and timing without sacrificing the ability to perform productive design-space exploration. Note that the MTL flow is fundamentally different from the approach of high-level synthesis (HLS). While HLS aims to take a high-level algorithm implementation in C or C++ and attempt to *automatically* infer a hardware implementation, the MTL methodology is a *manual* refinement process. The MTL methodology is orthogonal to HLS and allows a designer full control to progressively tune their model with respect to timing accuracy and implementation detail.

A key challenge of the MTL methodology is maintaining compatibility between FL, CL, and RTL models, which promotes both the reuse of test harnesses and the ability to co-simulate components at different abstraction levels. Integrating FL models into such a flow is particularly challeng-

26

ing due to their lack of timing information. In Chapter 3, I discuss how PyMTL, a Python-based implementation of an MTL methodology, addresses these challenges; PyMTL takes advantage of the mechanisms previously described in Section 2.2.4 to ease the process of vertically integrated hardware design.

To provide a brief preview of PyMTL: PyMTL is a unified, Python-based framework to facilitate a tightly integrated FL/CL/RTL methodology. PyMTL leveraging a concurrent-structural modeling approach, and naturally supports incremental refinement from high-level algorithms to cycle-approximate model to cycle-accurate implementation. An embedded DSL allows Python to be used as a hardware generation language for construction of highly-parameterized hardware designs. These parameterized designs can then be translated into synthesizable Verilog instances for use with commercial toolflows. Due to the unified nature of the framework, high-level cycle-approximate models can naturally be simulated alongside detailed RTL implementations, allowing users to finely control speed and accuracy tradeoffs on a module by module basis. Existing Verilog IP can be incorporated for co-simulation using a Verilator-based translation toolchain, enabling those with significant Verilog experience to leverage Python as a productive verification language. Finally, PyMTL provides several productivity components to ease the process of constructing and flexibly unit testing high-level models with latency-insensitive interfaces.

CHAPTER 3
PYMTL: A UNIFIED FRAMEWORK FOR MODELING TOWARDS LAYOUT

Given the numerous barriers involved in performing vertically integrated hardware design, I developed PyMTL[1] as a framework to help address many of the pain points encountered in computer architecture research. PyMTL began as a simple hardware-generation language in Python, an alternative to Verilog that would enable the construction of more parameterizable and reusable RTL models. However, it quickly became clear that there was considerable value in providing tight integration with more abstract cycle-level and functional-level models in PyMTL, and that Python's high-level, general-purpose programming facilities provided an incredible opportunity to improve the productivity of constructing FL and CL models as well. The implementation of PyMTL was heavily influenced by ideas introduced in prior work; a key lesson learned from my work on PyMTL is that the unification of these ideas into a single framework produces productivity benefits greater than the sum of its parts. In particular, combining embedded domain-specific languages (EDSL) with just-in-time (JIT) specializers is a particularly attractive and promising approach for constructing hardware modeling frameworks going forward.

This chapter discusses the features, design, and software architecture of the PyMTL framework. This includes the Python-based embedded-DSL used to describe PyMTL hardware models and the model/tool split implemented in the framework architecture to enable modularity and extensibility. The use of the PyMTL for implementing a vertically integrated, modeling towards layout design methodology is demonstrated through several simple examples. Finally, the SimJIT just-in-time specializer for generating optimized code from PyMTL embedded-DSL descriptions is introduced and evaluated as a technique for reducing the overhead of PyMTL simulations.

3.1 Introduction

Limitations in technology scaling have led to a growing interest in non-traditional system architectures that incorporate heterogeneity and specialization as a means to improve performance under strict power and energy constraints. Unfortunately, computer architects exploring these more exotic architectures generally lack existing physical designs to validate their power and performance

---

[1]PyMTL loosely stands for [Py]thon framework for [M]odeling [T]owards [L]ayout and is pronounced the same as "py-metal".

models. The lack of validated models makes it challenging to accurately evaluate the computational efficiency of these designs [BGOS12, GKO$^+$00, GKB09, GPD$^+$14, DBK01, CLSL02]. As specialized accelerators become more integral to achieving the performance and energy goals of future hardware, there is a crucial need for researchers to supplement cycle-level simulation with algorithmic exploration and RTL implementation.

Future computer architecture research will place an increased emphasis on a methodology we call *modeling towards layout* (MTL). While computer architects have long leveraged multiple modeling abstractions (functional level, cycle level, register-transfer level) to trade off simulation time and accuracy, an MTL methodology aims to vertically integrate these abstractions for iterative refinement of a design from algorithm, to exploration, to implementation. Although an MTL methodology is especially valuable for prototyping specialized accelerators and exploring more exotic architectures, it has general value as a methodology for more traditional architecture research as well.

Unfortunately, attempts to implement an MTL methodology using existing publicly-available research tools reveals numerous practical challenges we call the *computer architecture research methodology gap*. This gap is manifested as the distinct languages, design patterns, and tools commonly used by functional level (FL), cycle level (CL), and register-transfer level (RTL) modeling. The computer architecture research methodology gap exposes a critical need for a new vertically integrated framework to facilitate rapid design-space exploration and hardware implementation. Ideally such a framework would use a single specification language for FL, CL, and RTL modeling, enable multi-level simulations that mix models at different abstraction levels, and provide a path to design automation toolflows for extraction of credible area, energy, and timing results.

In this chapter, I introduce PyMTL, my attempt to construct such a unified, highly productive framework for FL, CL, and RTL modeling. PyMTL leverages a common high-productivity language (Python2.7) for behavioral specification, structural elaboration, and verification, enabling a rapid code-test-debug cycle for hardware modeling. Concurrent-structural modeling combined with latency-insensitive design allows reuse of test benches and components across abstraction levels while also enabling mixed simulation of FL, CL, and RTL models. A model/tool split provides separation of concerns between model specification and simulator generation letting architects focus on implementing hardware, not simulators. PyMTL's modular construction encourages extensibility: using elaborated model instances as input, users can write custom tools (also in Python)

such as simulators, translators, analyzers, and visualizers. Python's glue language facilities provide flexibility by allowing PyMTL models and tools to be extended with C/C++ components or embedded within existing C/C++ simulators [SIT+14]. Finally, PyMTL serves as a productive hardware generation language for building synthesizable hardware templates thanks to an HDL translation tool that converts PyMTL RTL models into Verilog-2001 source.

Leveraging Python as a modeling language improves model conciseness, clarity, and implementation time [Pre00, CNG+06], but comes at a significant cost to simulation time. For example, a pure Python cycle-level mesh network simulation in PyMTL exhibits a $300\times$ slowdown when compared to an identical simulation written in C++. To address this *performance-productivity gap*, inspiration is taken from the scientific computing community which has increasingly adopted *productivity-level languages* (e.g., MATLAB, Python) for computationally intensive tasks by replacing hand-written *efficiency-level language* code (e.g., C, C++) with dynamic techniques such as just-in-time (JIT) compilation [num14, RHWS12, CBHV10] and selective-embedded JIT specialization [CKL+09, BXF13].

I also introduce SimJIT, a custom just-in-time specializer that takes CL and RTL models written in PyMTL and automatically generates, compiles, links, and executes fast, Python-wrapped C++ code seamlessly within the PyMTL framework. SimJIT is both *selective* and *embedded* providing much of the benefits described in previous work on domain-specific embedded specialization [CKL+09]. SimJIT delivers significant speedups over CPython (up to $34\times$ for CL models and $63\times$ for RTL models), but sees even greater benefits when combined with PyPy, an interpreter for Python with a meta-tracing JIT compiler [BCFR09]. PyPy is able to optimize unspecialized Python code as well as hot paths between Python and C++, boosting the performance of SimJIT simulations by over $2\times$ and providing a net speedup of $72\times$ for CL models and $200\times$ for RTL models. These optimizations mitigate much of the performance loss incurred by using a productivity-level language, closing the performance gap between PyMTL and C++ simulations to within 4–$6\times$.

3.2 The Design of PyMTL

PyMTL is a proof-of-concept framework designed to provide a unified environment for constructing FL, CL, and RTL models. The PyMTL framework consists of a collection of classes implementing a concurrent-structural, embedded domain-specific language (EDSL) within Python

for hardware modeling, as well as a collection of tools for simulating and translating those models. The dynamic typing and reflection capabilities provided by Python enable succinct model descriptions, minimal boilerplate, and expression of flexible and highly parameterizable behavioral and structural components. The use of a popular, general-purpose programming language provides numerous benefits including access to mature numerical and algorithmic libraries, tools for test/development/debug, as well as access to the knowledge-base of a large, active development community.

The design of PyMTL was inspired by several mechanisms proposed in prior work. These mechanisms, previously discussed in detail in Section 2.2.4, are relisted here along with how they have explicitly influenced the design of PyMTL.

- **Concurrent-Structural Frameworks** – PyMTL is designed from the ground up to be a concurrent-structural framework. All communication between models occurs over port-based interfaces and all run-time model logic has concurrent execution semantics. This considerably simplifies the interfacing of FL, CL, and RTL models.

- **Unified Modeling Languages** – PyMTL utilizes a single specification language, Python2.7, to define all aspects of FL, CL, and RTL models. This includes model interfaces, structural connectivity, behavioral logic, static elaboration, and unit tests. In addition, PyMTL takes this concept one-step further by implementing the framework, simulation tool, translation tool, and user-defined extensions in Python2.7 as well.

- **Hardware Generation Languages** – A translation tool provided by the PyMTL framework in combination with the powerful static elaboration capabilities of PyMTL allows it to be used as a productive hardware generation language. PyMTL enables much more powerful parameterization and configuration facilities than traditional HDLs, while providing a path to EDA toolflows via translation of PyMTL RTL into Verilog.

- **HDL Integration** – RTL models written within PyMTL can be natively simulated alongside FL and CL models written in PyMTL. PyMTL can also automatically translate PyMTL RTL models into Verilog HDL and wrap them in a Python interface to co-simulate PyMTL-generated Verilog with pure-Python CL and FL models. In addition, PyMTL provides the capability to import hand-written Verilog IP for testing or co-simulation.

- **SEJITS** – Simulations in the PyMTL framework can optionally take advantage of SimJIT, a just-in-time specializer for CL and RTL models, to improve the performance of PyMTL simulations. While SimJIT-CL is a prototype implementation that only works for a small subset of models, SimJIT-RTL is a mature specializer in active use.

- **Latency-Insensitive Interfaces** – The PyMTL framework strongly encourages the use of latency-insensitive interfaces by providing a number of helper components for testing and creating models that use the `ValRdy` interface. These components include test sources and test sinks, port bundles that simplify the instantiation and connectivity of *ValRdy* interfaces, as well as adapters that expose user-friendly queue and list interfaces to hide the complexity of manually managing valid and ready signals.

3.3 PyMTL Models

PyMTL models are described in a concurrent-structural fashion: interfaces are port-based, logic is specified in concurrent logic blocks, and components are composed structurally. Users define model implementations as Python classes that inherit from `Model`. An example PyMTL class skeleton is shown in Figure 3.1. The `__init__` model constructor (lines 3–15) both executes *elaboration-time* configuration and declares *run-time* simulation logic. Elaboration-time configuration specializes model construction based on user-provided parameters. This includes the model interface (number, direction, message type of ports), internal constants, and structural hierarchy (wires, submodels, connectivity). Run-time simulation logic is defined using nested functions decorated with annotations that indicate their simulation-time execution behavior. Provided decorators include `@s.combinational` for combinational logic and `@s.tick_fl`, `@s.tick_cl`, and `@s.tick_rtl` for FL, CL, and RTL sequential logic, respectively. The semantics of signals (ports and wires) differ depending on whether they are updated in a combinational or sequential logic block. Signals updated in combinational blocks behave like wires; they are updated by writing their `.value` attributes and the concurrent block enclosing them only executes when its sensitivity list changes. Signals updated in sequential blocks behave like registers; they are updated by writing their `.next` attributes and the concurrent block enclosing them executes once every simulator cycle. Much like Verilog, submodel instantiation, structural connectivity, and behavioral logic definitions can be intermixed throughout the constructor.

32

```
1  class MyModel( Model ):
2
3    def __init__( s, constructor_params ):
4
5      # input  port  declarations
6      # output port  declarations
7      # other member declarations
8
9      # wire       declarations
10     # submodule declarations
11     # connectivity statements
12     # concurrent logic specification
13
14     # more connectivity statements
15     # more concurrent logic specification
```

Figure 3.1: PyMTL Model Template – A basic skeleton of a PyMTL model, which is simply a Python class that subclasses `Model`. Model classes are parameterized by arguments passed into the class initializer method, `__init__`. Parameterization arguments are used by statements inside the initializer method to perform *static-elaboration*: static, construction-time configuration of model attributes, connectivity, hierarchy, and even runtime behavior. Elaboration logic can mix wire and submodule declarations, structural connectivity, and concurrent logic definitions.

A few simple PyMTL model definitions are shown in Figure 3.2. The `Register` model consists of a constructor that declares a single input and output port (lines 12–13) as well as a sequential logic block using a `s.tick_rtl` decorated nested function (lines 19–21). Ports are parameterizable by message type, in this case a `Bits` fixed-bitwidth message of size `nbits` (line 11). Due to the pervasive use of the `Bits` message type in PyMTL RTL modeling, syntactic sugar has been added such that `InPort(4)` may be used in place of the more explicit `InPort(Bits(4))`. We use this shorthand for the remainder of our examples. The `Mux` model is parameterizable by bitwidth and number of ports: the input is declared as a list of ports using a custom shorthand provided by the PyMTL framework (line 11), while the select port bitwidth is calculated using a user-defined function `bw` (line 12). A single combinational logic block is defined during elaboration (lines 19–21). No explicit sensitivity list is necessary as this is automatically inferred during simulator construction. The `MuxReg` model structurally composes `Register` and `Mux` models by instantiating them like normal Python objects (lines 16–17) and connecting their ports via the `s.connect()` method (lines 21–25). Note that it is not necessary to declare temporary wires in order to connect submodules as ports can simply be directly connected. A *list comprehension* is used to instantiate the input port list of `MuxReg` (line 11). This Python idiom is commonly used in PyMTL design to flexibly construct parameterizable lists of ports, wires, and submodules.

The examples in Figure 3.2 also provide a sample of PyMTL models that are fully translatable to synthesizable Verilog HDL. Translatable models must: (1) describe all behavioral logic within within `s.tick_rtl` and `s.combinational` blocks, (2) use only a restricted, translatable subset of Python for logic within these blocks, and (3) pass all data using ports or wires with fixed-bitwidth

```
1  # A purely sequential model.
2  class Register( Model ):
3
4    # Initializer arguments specify the
5    # model is parameterized by bitwidth.
6    def __init__(s, nbits):
7
8      # Port interface declarations must
9      # specify both directionality and
10     # signal datatype of each port.
11     dtype = Bits( nbits )
12     s.in_ = InPort ( dtype )
13     s.out = OutPort( dtype )
14
15     # Concurrent blocks annotated with
16     # @tick* execute every clock cycle.
17     # Writes to .next have non-blocking
18     # update semantics.
19     @s.tick_rtl
20     def seq_logic():
21       s.out.next = s.in_
```

```
1  # A purely combinational model.
2  class Mux( Model ):
3
4    # Model is parameterized by bitwidth
5    # and number of input ports.
6    def __init__(s, nbits, nports):
7
8      # Integer nbits act as a shorthand
9      # for Bits( nbits ); concise array
10     # syntax declares an InPort list.
11     s.in_ = InPort[nports](nbits)
12     s.sel = InPort (bw(nports))
13     s.out = OutPort(nbits)
14
15     # @combinational annotated blocks
16     # only execute when input values
17     # change. Writes to .value have
18     # blocking update semantics.
19     @s.combinational
20     def comb_logic():
21       s.out.value = s.in_[s.sel]
```

```
1  # A purely structural model.
2  class MuxReg( Model ):
3
4    # The dtype = 8 default value, a shorthand for dtype = Bits(8), could
5    # alternatively receive a more complex user-defined type like a BitStruct.
6    def __init__( s, dtype = 8, nports = 4 ):
7
8      # The parameterized interface uses a user-defined bw() function to
9      # compute the correct bitwidth for sel. A list comprehension specifies
10     # the number of input ports, equivalent to InPort[nports](dtype).
11     s.in_ = [ InPort( dtype ) for x in range( nports ) ]
12     s.sel = InPort ( bw( nports ) )
13     s.out = OutPort( dtype )
14
15     # Parameterizable child models are instantiated like Python classes.
16     s.reg_ = Register( dtype )
17     s.mux  = Mux     ( dtype, nports )
18
19     # Input and output ports of the parent and child models are
20     # structurally connected using s.connect() statements.
21     s.connect( s.sel, s.mux.sel )
22     for i in range( nports ):
23       s.connect( s.in_[i], s.mux.in_[i] )
24     s.connect( s.mux.out,  s.reg_.in_ )
25     s.connect( s.reg_.out, s.out      )
```

Figure 3.2: PyMTL Example Models – Basic RTL models demonstrating sequential, combinational, and structural components in PyMTL. Powerful construction and elaboration logic enables design of highly-parameterizable models, while remaining Verilog-translatable.
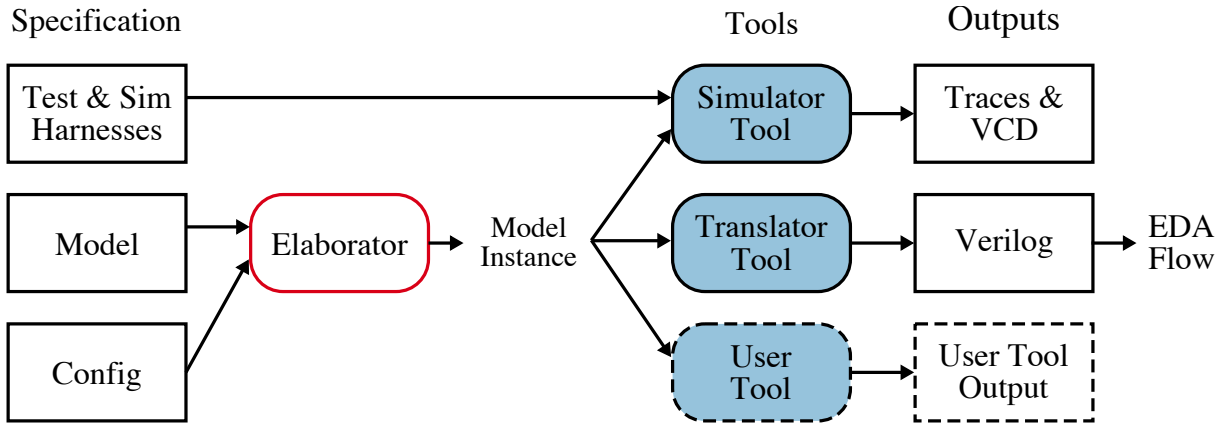
Figure 3.3: PyMTL Software Architecture – A model and configuration are elaborated into a model instance; tools manipulate the model instance to simulate or translate the design.

message types (like `Bits`). While these restrictions limit some of the expressive power of Python, PyMTL provides mechanisms such as `BitStructs`, `PortBundles`, and type inference of local temporaries to improve the succinctness and productivity of translatable RTL modeling in PyMTL. Purely structural models like `MuxReg` are always translatable if all child models are translatable. This enables the full power of Python to be used during elaboration. Even greater flexibility is provided to non-translatable FL and CL models as they may contain arbitrary Python code within their `@s.tick_fl` and `@s.tick_cl` behavioral blocks. Examples of FL and CL models, shown in Figures 3.7, 3.8, and 3.11, will be discussed in further detail in Sections 3.5.1 and 3.5.2.

3.4 PyMTL Tools

The software architecture of the PyMTL framework is shown in Figure 3.3. User-defined models are combined with their configuration parameters to construct and elaborate model classes into model instances. Model instances act as in-memory representations of an elaborated design that can be accessed, inspected, and manipulated by various tools, just like a normal Python object. For example, the `TranslationTool` takes PyMTL RTL models, like those in Figure 3.2, inspects their structural hierarchy, connectivity, and concurrent logic, then uses this information to generate synthesizable Verilog that can be passed to an electronic design automation (EDA) toolflow. Similarly, the `SimulationTool` inspects elaborated models to automatically register concurrent logic blocks, detect sensitivity lists, and analyze the structure of connected ports to generate op-

35

```
1  @pytest.mark.parametrize(
2    'nbits,nports', [( 8, 2), ( 8, 3), ( 8, 4), ( 8, 8)
3                      (16, 2), (16, 3), (16, 4), (16, 8),
4                      (32, 2), (32, 3), (32, 4), (32, 8)]
5  )
6  def test_muxreg( nbits, nports, test_verilog ):
7    model = MuxReg( nbits, nports )
8    model.elaborate()
9    if test_verilog:
10     model = TranslationTool( model )
11
12   sim = SimulationTool( model )
13   for inputs, sel, output in gen_vectors(nbits,nports):
14     for i, val in enumerate( inputs ):
15       model.in_[i].value = val
16     model.sel.value = sel
17     sim.cycle()
18     assert model.out == output
```

Figure 3.4: PyMTL Test Harness – The `SimulationTool` and `py.test` package are used to simulate and verify the `MuxReg` module in Figure 3.2. A command-line flag uses the `TranslationTool` to automatically convert the `MuxReg` model into Verilog and test it within the same harness.

timized Python simulators. The modular nature of this model/tool split encourages extensibility making it easy for users to write their own custom tools such as linters, translators, and visualization tools. More importantly, it provides a clean boundary between hardware modeling logic and simulator implementation logic letting users focus on hardware design rather than simulator software engineering.

The PyMTL framework uses the open-source testing package `py.test` [pyt14a] along with the provided `SimulationTool` to easily create extensive unit-test suites for each model. One such unit-test can be seen in Figure 3.4. After instantiating and elaborating a PyMTL model (lines 7–8), the test bench constructs a simulator using the `SimulationTool` (line 12) and tests the design by setting input vectors, cycling the simulator, and asserting outputs (lines 14–18). A number of powerful features are demonstrated in this example: the `py.test` `@parametrize` decorator instantiates a large number of test configurations from a single test definition (lines 1–5), user functions are used to generate configuration-specific test vectors (line 13), and the model can be automatically translated into Verilog and verified within the same test bench by simply passing the `--test-verilog` flag at the command line (lines 9–10). In addition, `py.test` can provide test

coverage statistics and parallel test execution on multiple cores or multiple machines by importing additional `py.test` plugins [pyt14b, pyt14c].

3.5 PyMTL By Example

In this section, I demonstrate how PyMTL can be used to model, evaluate, and implement two models for architectural design-space exploration: an accelerator coprocessor and an on-chip network. Computer architects are rarely concerned only with the performance of a single component, rather we aim to determine how a given mechanism may impact system performance as a whole. With this in mind, the accelerator is implemented in the context of the hypothetical heterogeneous system shown in Figure 3.5a. This system consists of numerous compute tiles interconnected by an on-chip network. Section 3.5.1 will explore the implementation of an accelerator on a single tile, while Section 3.5.2 will investigate a simple mesh network that might interconnect such tiles.

3.5.1 Accelerator Coprocessor

This section describes the modeling of a dot-product accelerator within the context of a single tile containing a simple RISC processor, an L1 instruction cache, and an L1 data cache. The dot-product operator multiplies and accumulates the values of two equal-length vectors, returning a single number. This accelerator is implemented as a coprocessor with several configuration registers to specify the size of the two input vectors, their base addresses, and a start command. The coprocessor is designed to share a port to the L1 data cache with the processor as shown in Figure 3.5a. A *modeling towards layout* methodology is used to refine the accelerator design from algorithm to implementation by first constructing a functional-level model, and gradually refining it into cycle-level and finally register-transfer-level models.

Functional Level Architects build an FL model as a first step in the design process to familiarize themselves with an algorithm and create a golden model for validating more detailed implementations. Figure 3.6 demonstrates two basic approaches to constructing a simple FL model. The first approach (lines 1–2) manually implements the dot-product algorithm in Python. This approach provides an opportunity for the designer to rapidly experiment with alternative algorithm implementations. The second approach (lines 4–5) simply calls the `dot` library function provided by

(a) Block Diagram

(b) Post-Place-and-Route Layout
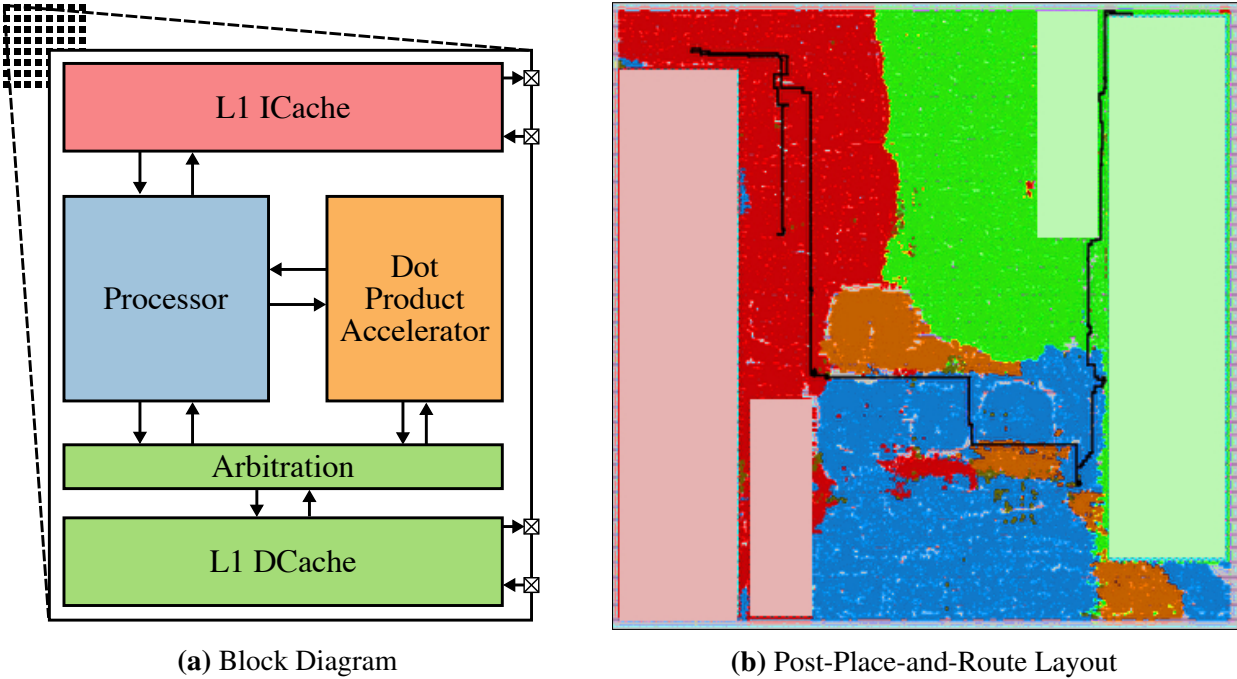
Figure 3.5: Hypothetical Heterogeneous Architecture – (a) Accelerator-augmented compute tiles interconnected by an on-chip network; (b) Synthesized, placed, and routed layout of compute tile shown in (a). Processor, cache, and accelerator RTL for this design were implemented and tested entirely in PyMTL, automatically translated into Verilog HDL, then passed to a Synopsys toolflow. Processor shown in blue, accelerator in orange, L1 caches in red and green, and critical path in black.

the numerical package NumPy [Oli07]. This approach provides immediate access to a verified, optimized, high-performance golden reference.

Unfortunately, integrating such FL implementations into a computer architecture simulation framework can be a challenge. Our accelerator is designed as a coprocessor that interacts with both a processor and memory, so the FL model must implement communication protocols to interact with the rest of the system. This is a classic example of the methodology gap.

Figure 3.7 demonstrates a PyMTL FL model for the dot-product accelerator capable of interacting with FL, CL, and RTL models of the processor and memory. While more verbose than the simple implementations in Figure 3.6, the `DotProductFL` model must additionally control interactions with the processor, memory, and accelerator state. This additional complexity is greatly simplified by several PyMTL provided components: `ReqRespBundles` encapsulate collections of signals needed for latency-insensitive communication with the processor and memory (lines 3–4), the `ChildReqRespQueueAdapter` provides a simple queue-based interface to the

38

```
1  def dot_product_manual( src0, src1 ):
2    return sum( [x*y for x,y in zip(src0, src1)] )
3
4  def dot_product_library( src0, src1 ):
5    return numpy.dot( src0, src1 )
```

Figure 3.6: Functional Dot Product Implementation – A functional implementation of the dot product operator. Both a manual implementation in Python and a higher-performance library implementation are shown.

```
1  class DotProductFL( Model ):
2    def __init__( s, mem_ifc_types, cpu_ifc_types ):
3      s.cpu_ifc = ChildReqRespBundle ( cpu_ifc_types )
4      s.mem_ifc = ParentReqRespBundle ( mem_ifc_types )
5
6      s.cpu  = ChildReqRespQueueAdapter( s.cpu_ifc )
7      s.src0 = ListMemPortAdapter      ( s.mem_ifc )
8      s.src1 = ListMemPortAdapter      ( s.mem_ifc )
9
10     @s.tick_fl
11     def logic():
12       s.cpu.xtick()
13       if not s.cpu.req_q.empty() and not s.cpu.resp_q.full():
14         req = s.cpu.get_req()
15         if req.ctrl_msg == 1:
16           s.src0.set_size( req.data )
17           s.src1.set_size( req.data )
18         elif req.ctrl_msg == 2: s.src0.set_base( req.data )
19         elif req.ctrl_msg == 3: s.src1.set_base( req.data )
20         elif req.ctrl_msg == 0:
21           result = numpy.dot( s.src0, s.src1 )
22           s.cpu.push_resp( result )
```

Figure 3.7: PyMTL DotProductFL Accelerator – Concurrent-structural modeling allows composition of FL models with CL and RTL models, but introduces the need to implement communication protocols. `QueueAdapter` and `ListAdapter` proxies provide programmer-friendly, method-based interfaces that hide the complexities of these protocols.

`ChildReqRespBundle` and automatically manages latency-insensitive communication to the processor (lines 6, 12–14, 22), and the `ListMemPortAdapter` provides a list-like interface to the `ParentReqRespBundle` and automatically manages the latency-insensitive communication to the memory (lines 7–8).

Of particular note is the `ListMemPortAdapter` which allows us to reuse `numpy.dot` from Figure 3.6 without modification. This is made possible by the greenlets concurrency package [pyt14d] that enables proxying array index accesses into memory request and response transactions

over the latency-insensitive, port-based model interfaces. These proxies facilitate the composition of existing, library-provided utility functions with port-based processors and memories to quickly create a target for validation and software co-development.

Cycle Level   Construction of a cycle-level model provides a sense of the timing behavior of a component, enabling architects to estimate system-level performance and make first-order design decisions prior to building a detailed RTL implementation. Figure 3.8 shows an implementation of the `DotProductCL` model in PyMTL. Rather than faithfully emulating detailed pipeline behavior, this model simply aims to issue and receive memory requests in a cycle-approximate manner by implementing a simple pipelining scheme. Like the FL model, the CL model takes advantage of higher-level PyMTL library constructs such as the `ReqResponseBundles` and `QueueAdapters` to simplify the design, particularly with regards to interfacing with external communication protocols (lines 3–7). Logic is simplified by pre-generating all memory requests and storing them in a list once the go signal is set (line 39), this list is used to issue requests to memory as backpressure allows (lines 23–24). Data is received from the memory in a pipelined manner and stored in another list (lines 25–26). Once all data is received it is separated, passed into `numpy.dot`, and returned to the processor (lines 28–31).

Because `DotProductCL` exposes an identical port-based interface to `DotProductFL`, construction of the larger tile can be created using an incremental approach. These steps include writing unit-tests based on golden FL model behavior, structurally composing the FL model with the processor and memory to validate correct system behavior, verifying the CL model in isolation by reusing the FL unit tests, and finally swapping the FL model and the CL model for final system-level integration testing. This pervasive testing gives us confidence in our model, and the final composition of the CL accelerator with CL or RTL memory and processor models allow us to evaluate system-level behavior.

To estimate the performance impact of our accelerator, a more detailed version of `DotProductCL` is combined with CL processor and cache components to create a CL tile. This accelerator-augmented tile is used to execute a 1024×1024 matrix-vector multiplication kernel (a computation consisting of 1024 dot products). The resulting CL simulation estimates our accelerator will provide a $2.9\times$ speedup over a traditional scalar implementation with loop-unrolling optimizations.

```
1  class DotProductCL( Model ):
2    def __init__( s, mem_ifc_types, cpu_ifc_types ):
3      s.cpu_ifc = ChildReqRespBundle ( cpu_ifc_types )
4      s.mem_ifc = ParentReqRespBundle( mem_ifc_types )
5
6      s.cpu     = ChildReqRespQueueAdapter ( s.cpu_ifc )
7      s.mem     = ParentReqRespQueueAdapter( s.mem_ifc )
8
9      s.go    = False
10     s.size  = 0
11     s.src0  = 0
12     s.src1  = 0
13     s.data  = []
14     s.addrs = []
15
16     @s.tick_cl
17     def logic():
18       s.cpu.xtick()
19       s.mem.xtick()
20
21       if s.go:
22
23         if s.addrs and not s.mem.req_q.full():
24           s.mem.push_req( mreq( s.addrs.pop() ) )
25         if not s.mem.resp_q.empty():
26           s.data.append( s.mem.get_resp() )
27
28         if len( s.data ) == s.size*2:
29           result = numpy.dot( s.data[0::2], s.data[1::2] )
30           s.cpu.push_resp( result )
31           s.go = False
32
33       elif not s.cpu.req_q.empty() and not s.cpu.resp_q.full():
34         req = s.cpu.get_req()
35         if   req.ctrl_msg == 1: s.size = req.data
36         elif req.ctrl_msg == 2: s.src0 = req.data
37         elif req.ctrl_msg == 3: s.src1 = req.data
38         elif req.ctrl_msg == 0:
39           s.addrs = gen_addresses( s.size, s.src0, s.src1 )
40           s.go    = True
```

Figure 3.8: PyMTL DotProductCL Accelerator – Python's high-level language features are used to quickly prototype a cycle-approximate model with pipelined memory requests. QueueAdapters wrap externally visible ports with a more user-friendly, queue-like abstraction for enqueuing and dequeuing data. These adapters automatically manage valid and ready signals of the latency insensitive interface and provide backpressure that is used to cleanly implement stall logic. The user-defined gen_addresses() function creates a list of memory addresses that are used to fetch data needed for the dot product operation; once all this input data has been fetched numpy.dot() is used to compute the result. A pipeline component could be added to delay the return of the dot product computation and more realistically model the timing behavior of the hardware target.

Configuration	CL Speedup (Cycles)	RTL Speedup (Cycles)	Cycle Time	Area	Execution Time
Proc+Cache	1.00×	1.00×	2.10ns	1.04mm^2	9.3ms
Proc+Cache+Accel	2.90×	2.88×	2.21ns	1.06mm^2	3.4ms

Table 3.1: DotProduct Coprocessor Performance – Performance comparison for tile in Figure 3.5a with the accelerator coprocessor (Proc+Cache+Accel) and without (Proc+Cache); both configurations include an RTL implementations of a 5-stage RISC processor, instruction cache, and data cache. Performance estimates of execution cycles generated from the CL model (CL Speedup) are quite close to the RTL implementation (RTL speedup): 2.90× estimated versus 2.88× actual.

Register-Transfer Level   The CL model allowed us to quickly obtain a cycle-approximate performance estimate for our accelerator-enhanced tile in terms of *simulated cycles*, however, *area*, *energy*, and *cycle time* are equally important metrics that must also be considered. Unfortunately, accurately predicting these metrics from high-level models is notoriously difficult. An alternative approach is to use an industrial EDA toolflow to extract estimates from a detailed RTL implementation. Building RTL is often the most appropriate approach for obtaining credible metrics, particularly when constructing exotic accelerator architectures.

PyMTL attempts to address many of the challenges associated with RTL design by providing a productive environment for constructing highly parameterizable RTL implementations. Figures 3.9 and 3.10 show the top-level and datapath code for the `DotProductRTL` model. The PyMTL EDSL provides a familiar Verilog-inspired syntax for traditional combinational and sequential bit-level design using the `Bits` data-type, but also layers more advanced constructs and powerful elaboration-time capabilities to improve code clarity. A concise top-level module definition is made possible by the use of `PortBundles` and the `connect_auto` method, which automatically connects parent and child signals based on signal name (lines 1-8). `BitStructs` are used as message types to connect control and status signals (lines 14–15), improving code clarity by providing named access to bitfields (lines 28, 34–35, 39). Mixing of wire declarations, sequential logic definitions, combinational logic definitions, and parameterizable submodule instantiations (lines 58–61) enable code arrangements that clearly demarcate pipeline stages. In addition, `DotProductRTL` shares the same parameterizable interface as the FL and CL models enabling reuse of unmodified FL and CL test benches for RTL validation before automatic translation into synthesizable Verilog.

```
 1  class DotProductRTL( Model ):
 2    def __init__( s, mem_ifc_types, cpu_ifc_types ):
 3      s.cpu_ifc = ChildReqRespBundle ( cpu_ifc_types )
 4      s.mem_ifc = ParentReqRespBundle( mem_ifc_types )
 5
 6      s.dpath = DotProductDpath( mem_ifc_types, cpu_ifc_types )
 7      s.ctrl  = DotProductCtrl ( mem_ifc_types, cpu_ifc_types )
 8      s.connect_auto( s.dpath, s.ctrl )
 9
10  class DotProductDpath( Model ):
11    def __init__( s, mem_ifc_types, cpu_ifc_types ):
12      s.cpu_ifc = ChildReqRespBundle ( cpu_ifc_types )
13      s.mem_ifc = ParentReqRespBundle( mem_ifc_types )
14      s.cs       = InPort ( CtrlSignals()   )
15      s.ss       = OutPort( StatusSignals() )
16
17      #--- Stage M: Memory Request -----------------------------
18      s.count      = Wire( cpu_ifc_types.req .data.nbits )
19      s.size       = Wire( cpu_ifc_types.req .data.nbits )
20      s.src0_addr_M = Wire( mem_ifc_types.req .addr.nbits )
21      s.src1_addr_M = Wire( mem_ifc_types.req .addr.nbits )
22
23      @s.tick_rtl
24      def stage_seq_M():
25        ctrl_msg = s.cpu_ifc.req_msg .ctrl_msg
26        cpu_data = s.cpu_ifc.req_msg .data
27
28        if s.cs.update_M:
29          if   ctrl_msg == 1: s.size        .next = cpu_data
30          elif ctrl_msg == 2: s.src0_addr_M.next = cpu_data
31          elif ctrl_msg == 3: s.src1_addr_M.next = cpu_data
32          elif ctrl_msg == 0: s.ss.go       .next = True
33
34        if   s.cs.count_clear_M: s.count.next = 0
35        elif s.cs.count_en_M:    s.count.next = s.count + 1
36
37      @s.combinational
38      def stage_comb_M():
39        if s.cs.baddr_sel_M == src0: base_addr_M = s.src0_addr_M
40        else:                        base_addr_M = s.src1_addr_M
41
42        s.mem_ifc.req_msg.type_.value = 0
43        s.mem_ifc.req_msg.addr.value = base_addr_M + (s.count<<2)
44
45        s.ss.last_item_M.value = s.count == (s.size - 1)
```

Figure 3.9: PyMTL DotProductRTL Accelerator – RTL implementation of a dot product accelerator in PyMTL, control logic is not shown for brevity. The PyMTL EDSL combines familiar HDL syntax with powerful elaboration capabilities for constructing parameterizable and Verilog-translatable models. The top-level interface used for DotProductRTL matches the interfaces used by DotProductFL and DotProductCL which, along with their implementations of the ValRdy latency-insensitive communication protocol, allow them to be drop-in replacements for each other.

43

```
46      #--- Stage R: Memory Response -----------------------------
47      s.src0_data_R = Wire( mem_ifc_types.resp.data.nbits )
48      s.src1_data_R = Wire( mem_ifc_types.resp.data.nbits )
49
50      @s.tick_rtl
51      def stage_seq_R():
52        mem_data = s.mem_ifc.resp_msg.data
53        if s.cs.src0_en_R: s.src0_data_R.next = mem_data
54        if s.cs.src1_en_R: s.src1_data_R.next = mem_data
55
56      #--- Stage X: Execute Multiply ----------------------------
57      s.result_X = Wire( cpu_ifc_types.req.data.nbits )
58      s.mul      = IntPipelinedMultiplier(
59                     nbits   = cpu_ifc_types.req.data.nbits,
60                     nstages = 4,
61                   )
62      s.connect_dict( { s.mul.op_a     : s.src0_data_R,
63                        s.mul.op_b     : s.src1_data_R,
64                        s.mul.product : s.result_X     } )
65
66      #--- Stage A: Accumulate ----------------------------------
67      s.accum_A   = Wire( cpu_ifc_types.resp.data.nbits )
68      s.accum_out = Wire( cpu_ifc_types.resp.data.nbits )
69
70      @s.tick_rtl
71      def stage_seq_A():
72        if    s.reset or s.cs.accum_clear_A:
73          s.accum_A.next = 0
74        elif s.cs.accum_en_A:
75          s.accum_A.next = s.accum_out
76
77      @s.combinational
78      def stage_comb_A():
79        s.accum_out.value = s.result_X + s.accum_A
80        s.cpu_ifc.resp_msg.value = s.accum_A
```

Figure 3.10: PyMTL DotProductRTL Accelerator Continued – Elaboration logic within `__init__()` mixes sequential/combinational blocks, wire declarations, module instantiations, and connectivity statements. Lines 58–64 instantiate and structurally connect a pipelined multiplier.

Figure 3.5b shows a synthesized, placed, and routed implementation of the tile in Figure 3.5a, including a 5-stage RISC processor, dot-product accelerator, instruction cache, and data cache. The entire tile was implemented, simulated, and verified in PyMTL before being translated into Verilog and passed to a Synopsys EDA toolflow. Using this placed-and-routed design, area, energy, and timing metrics were extracted for the tile. Execution performance of the RTL model and physical metrics for the placed-and-routed design can be seen in Table 3.1. The simulated cycles of the RTL implementation demonstrates a speedup of $2.88\times$, indicating that our CL model did a

particularly good job of modeling our implementation; this becomes more difficult to achieve with more complicated designs. The dot-product accelerator added an area overhead of 4% ($0.02\,\mathrm{mm}^2$) and increased the cycle time of the tile by approximately 5%. Fortunately, the improvement in simulated cycles resulted in a net execution time speedup of $2.74\times$. This performance improvement must be weighed against the overheads and the fact that the accelerator is only useful for dot-product computations.

3.5.2   Mesh Network

The previous section evaluated adding an accelerator to a single tile from Figure 3.5a in isolation, however, this tile is just one component in a much larger multi-tile system. In this section, the design and performance of a simple mesh network to interconnect these tiles is briefly explored.

Functional Level   Verifying tile behavior in the context of a multi-tile system can be greatly simplified by starting with an FL network implementation. PyMTL's concurrent-structural modeling approach allows us to quickly write a port-based FL model of our mesh network (behaviorally equivalent to a "magic" single-cycle crossbar) and connect it with FL, CL, or RTL tiles in order to verify our tiles and to provide a platform for multi-tile software development. Figure 3.11 shows a full PyMTL implementation of an FL network. PyMTL does not force the user to use the higher-level interface proxies utilized by the `DotProductFL` model; the `NetworkFL` model instead manually sets signals to implement the latency-insensitive communication protocol (lines 30–38). This provides users a fine granularity of control depending on preferences and modeling needs.

Cycle Level   A CL mesh network emulating realistic network behavior is implemented to investigate network performance characteristics. Figure 3.12 contains PyMTL code for a structural mesh network. This model is designed to take a PyMTL router implementation as a parameter (line 2) and structurally compose instances of this router into a complete network (lines 16–37). This approach is particularly powerful as it allows us to easily instantiate the network with either FL, CL, or RTL router models to trade-off accuracy and simulation speed. Alternatively, it allows us to quickly swap out routers with different microarchitectures for either verification or evaluation purposes. To construct a simple CL network model, we use `MeshNetworkStructural` to construct an 8x8 mesh network composed of routers using XY-dimension ordered routing and elastic-buffer

```
1  class NetworkFL( Model ):
2    def __init__( s, nrouters, nmsgs, data_nbits, nentries ):
3
4      # ensure nrouters is a perfect square
5      assert sqrt( nrouters ) % 1 == 0
6
7      net_msg = NetMsg( nrouters, nmsgs, data_nbits )
8      s.in_   = InValRdyBundle [ nrouters ]( net_msg )
9      s.out   = OutValRdyBundle[ nrouters ]( net_msg )
10
11     s.nentries     = nentries
12     s.output_fifos = [deque() for x in range(nrouters)]
13
14     @s.tick_fl
15     def network_logic():
16
17       # dequeue logic
18       for i, outport in enumerate( s.out ):
19         if outport.val and outport.rdy:
20           s.output_fifos[ i ].popleft()
21
22       # enqueue logic
23       for inport in s.in_:
24         if inport.val and inport.rdy:
25           dest = inport.msg.dest
26           msg  = inport.msg[:]
27           s.output_fifos[ dest ].append( msg )
28
29       # set output signals
30       for i, fifo in enumerate( s.output_fifos ):
31
32         is_full  = len( fifo ) == s.nentries
33         is_empty = len( fifo ) == 0
34
35         s.out[ i ].val.next = not is_empty
36         s.in_[ i ].rdy.next = not is_full
37         if not is_empty:
38           s.out[ i ].msg.next = fifo[ 0 ]
```

Figure 3.11: PyMTL FL Mesh Network – Functional-level model emulates the functionality but not the timing of a mesh network. This is behaviorally equivalent to an ideal crossbar. Resource constraints exist only on the model interface: multiple packets can enter the same queue in a single cycle, but only one packet may leave per cycle. Unlike the dot product FL model shown in Figure 3.7, this FL model does not use interface proxies and instead manually handles the signalling logic of the ValRdy protocol. The collections.deque class provided by the Python standard library is used to greatly simplify the implementation of queuing logic.

```
1  class MeshNetworkStructural( Model ):
2    def __init__( s, RouterType, nrouters, nmsgs, data_nbits, nentries ):
3
4      # ensure nrouters is a perfect square
5      assert sqrt( nrouters ) % 1 == 0
6
7      s.RouterType = RouterType
8      s.nrouters   = nrouters
9      s.params     = [nrouters, nmsgs, data_nbits, nentries]
10
11     net_msg = NetMsg( nrouters, nmsgs, data_nbits )
12     s.in_    = InValRdyBundle [ nrouters ]( net_msg )
13     s.out    = OutValRdyBundle[ nrouters ]( net_msg )
14
15     # instantiate routers
16     R = s.RouterType
17     s.routers = [ R(x, *s.params) for x in range(s.nrouters) ]
18
19     # connect injection terminals
20     for i in xrange( s.nrouters ):
21       s.connect( s.in_[i],  s.routers[i].in_[R.TERM] )
22       s.connect( s.out[i],  s.routers[i].out[R.TERM] )
23
24     # connect mesh routers
25     nrouters_1D = int( sqrt( s.nrouters ) )
26     for j in range( nrouters_1D ):
27       for i in range( nrouters_1D ):
28         idx = i + j * nrouters_1D
29         cur = s.routers[idx]
30         if i + 1 < nrouters_1D:
31           east = s.routers[ idx + 1 ]
32           s.connect(cur.out[R.EAST], east.in_[R.WEST])
33           s.connect(cur.in_[R.EAST], east.out[R.WEST])
34         if j + 1 < nrouters_1D:
35           south = s.routers[ idx + nrouters_1D ]
36           s.connect(cur.out[R.SOUTH], south.in_[R.NORTH])
37           s.connect(cur.in_[R.SOUTH], south.out[R.NORTH])
```

Figure 3.12: PyMTL Structural Mesh Network – Structurally composed network parameterized by network message type, network size, router buffering, and router type. Note that the router type parameter takes a Model class as input, which could potentially be either a PyMTL FL, CL, or RTL model depending on the desired simulation speed and accuracy characteristics. This demonstrates the power of static elaboration for creating highly-parameterizable models and model generators. This elaboration logic can use arbitrary Python while still remaining Verilog translatable as long as the user-provided RouterType parameter is a translatable RTL model. The use of ValRdyBundles significantly reduces structural connectivity complexity.
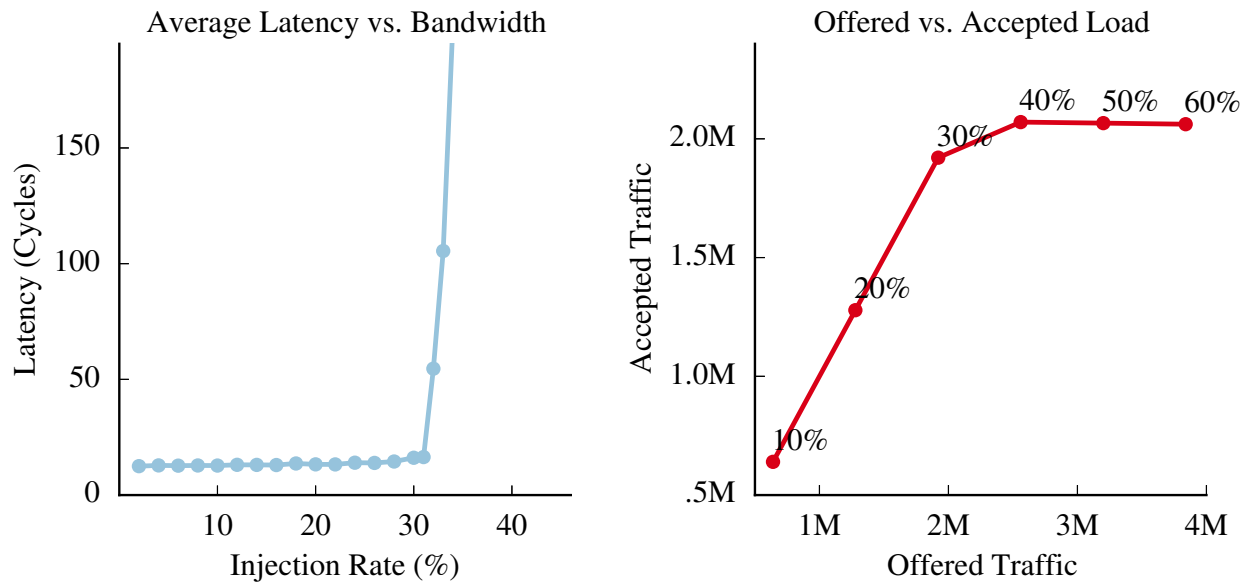
Figure 3.13: Performance of an 8x8 Mesh Network – Latency-bandwidth and offered-accepted traffic performance for a 64-node, XY-dimension ordered mesh network with on-off flow control. These plots, generated from simulations of a PyMTL cycle-level model, indicate saturation of the network is reached shortly after an injection rate of approximately 30%.

flow control. Simulations of this model allows us to quickly generate the network performance plots shown in Figure 3.13. These simulations estimate that the network has a zero-load latency of 13 cycles and saturates at an injection rate of 32%.

Register-Transfer Level Depending on our design goals, we may want to estimate area, energy, and timing for a single router, the entire network in isolation, or the network with the tiles attached. An RTL network can be created using the same top-level structural code as in Figure 3.12 by simply passing in an RTL router implementation as a parameter. Structural code in PyMTL is always Verilog translatable as long as all leaf modules are also Verilog translatable.

3.6   SimJIT: Closing the Performance-Productivity Gap

While the dynamic nature of Python greatly improves the expressiveness, productivity, and flexibility of model code, it significantly degrades simulation performance when compared to a statically compiled language like C++. We address this performance limitation by using a hybrid just-in-time optimization approach. We combine SimJIT, a custom just-in-time specializer for

converting PyMTL models into optimized C++ code, with the PyPy meta-tracing JIT interpreter. Below we discuss the design of SimJIT and evaluate its performance on CL and RTL models.

3.6.1   SimJIT Design

SimJIT consists of two distinct specializers: SimJIT-CL for specializing cycle-level PyMTL models and SimJIT-RTL for specializing register-transfer-level PyMTL models. Figure 3.14 shows the software architecture of the SimJIT-CL and SimJIT-RTL specializers. Currently, the designer must manually invoke these specializers on their models, although future work could consider adding support to automatically traverse the model hierarchy to find and specialize appropriate CL and RTL models.

SimJIT-CL begins with an elaborated PyMTL model instance and uses Python's reflection capabilities to inspect the model's structural connectivity and concurrent logic blocks. We are able to reuse several model optimization utilities from the previously described `SimulationTool` to help in generating optimized C++ components. We also leverage the `ast` package provided by the Python Standard Library to implement translation of concurrent logic blocks into C++ functions. The translator produces both C++ source implementing the optimized model as well as a C interface wrapper so that this C++ source may be accessed via CFFI, a fast foreign function interface library for calling C code from Python. Once code generation is complete, it is automatically compiled into a C shared library using LLVM, then imported into Python using an automatically generated PyMTL wrapper. This process gives the library a port-based interface so that it appears as a normal PyMTL model to the user.

Similar to SimJIT-CL, the SimJIT-RTL specializer takes an elaborated PyMTL model instance and inspects it to begin the translation process. Unlike SimJIT-CL, SimJIT-RTL does not attempt to perform any optimizations, rather it directly translates the design into equivalent synthesizable Verilog HDL. This translated Verilog is passed to Verilator, an open-source tool for generating optimized C++ simulators from Verilog source [ver13]. We combine the verilated C++ source with a generated C interface wrapper, compile it into a C shared library, and once again wrap this in a generated PyMTL model.

While both SimJIT-CL and SimJIT-RTL can generate fast C++ components that significantly improve simulation time, the Python interface still has a considerable impact on simulation performance. We leverage PyPy to optimize the Python simulation loop as well as the hot-paths between

49
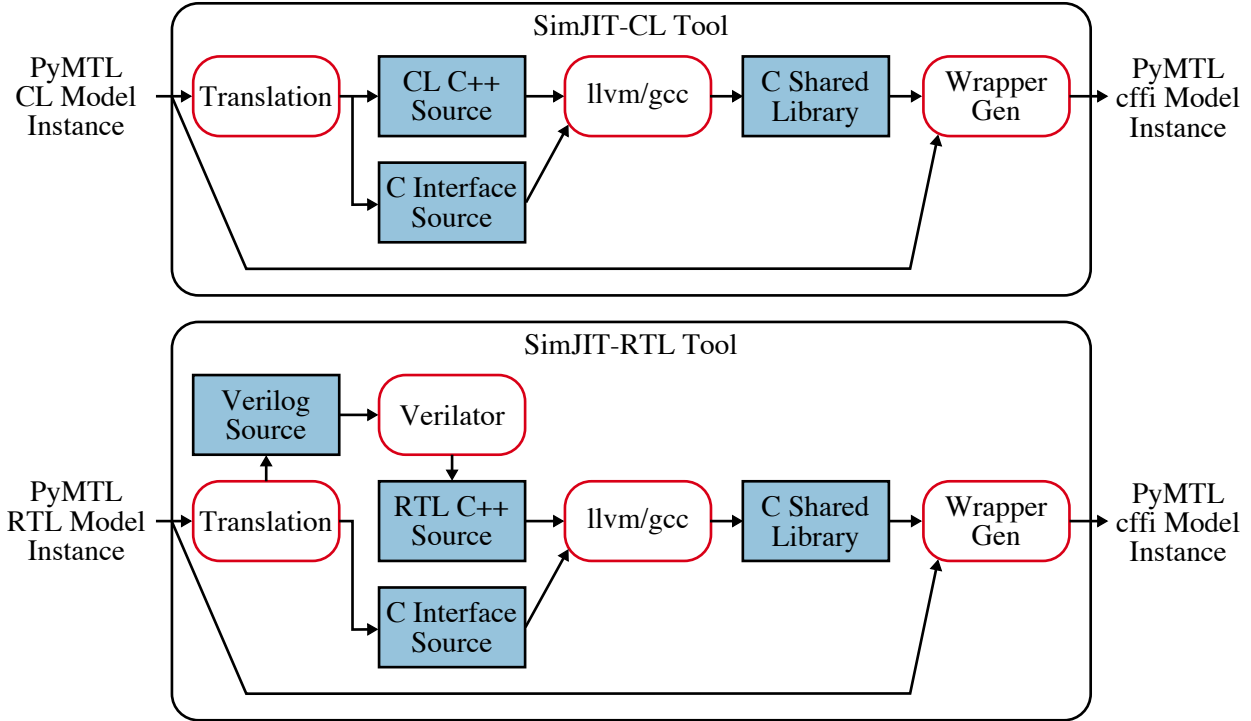
Figure 3.14: SimJIT Software Architecture – SimJIT consists of two specializers: one for CL models and one for RTL models. Each specializer can automatically translate PyMTL models into C++ and generate the appropriate wrappers to enable these C++ implementations to appear as standard PyMTL models.

the Python and C++ call interface, significantly reducing the overhead of using Python component wrappers. Compilation time of the specializer can also take a considerable amount of time, especially for SimJIT-RTL. For this reason, PyMTL includes support for automatically caching the results from translation for SimJIT-RTL. While not currently implemented, caching the results from translation for SimJIT-CL should be relatively straight-forward. In the next two sections, we examine the performance benefits of SimJIT and PyPy in greater detail, using the PyMTL models discussed in Sections 3.5.1 and 3.5.2 as examples.

3.6.2 SimJIT Performance: Accelerator Tile

We construct 27 different tile models at varying levels of detail by composing FL, CL, and RTL implementations of the processor (P), caches (C), and accelerator (A) for the compute tile in Figure 3.5a. Each configuration is described as a tuple $\langle P, C, A \rangle$ where each entry is FL, CL, or RTL. Each configuration is simulated in CPython with no optimizations and also simulated again using both SimJIT and PyPy. For this experiment, a slightly more complicated dot product
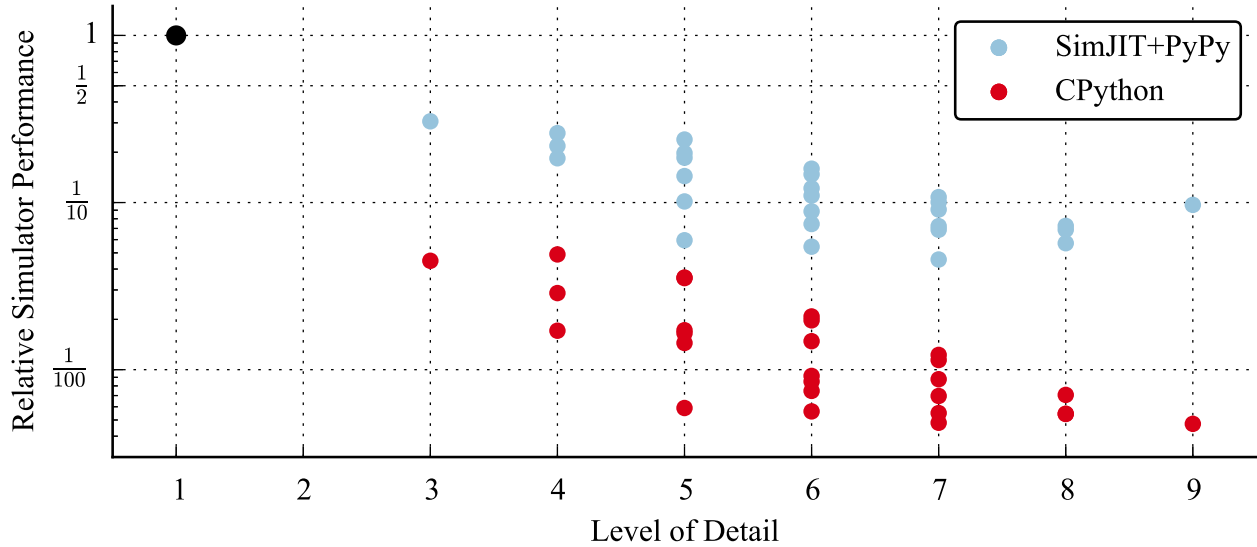
Figure 3.15: Simulator Performance vs. Level of Detail – Simulator performance using CPython and SimJIT+PyPy with the processor, memory, and accelerator modeled at various levels of abstraction. Results are normalized against the pure ISA simulator using PyPy. Level of detail (LOD) is measured by allocating a score of one for FL, two for CL, and three for RTL and then summing across the three models. For example, a FL processor composed with a CL memory system and RTL accelerator would have an LOD of 1+2+3 = 6.

accelerator was used than the one described in Section 3.5.1. SimJIT+PyPy runs applied SimJIT-RTL specialization to all RTL components in a model, whereas SimJIT-CL optimizations were only applied to the caches due to limitations of our current proof-of-concept SimJIT-CL specializer. Figure 3.15 shows the simulation performance of each run plotted against a "level of detail" (LOD) score assigned to each configuration. LOD is calculated such that LOD $= p + c + a$ where p, c, and a have a value corresponding to the model complexity: FL $= 1$, CL $= 2$, RTL $= 3$. Note that the LOD metric is not meant to be an exact measure of model accuracy but rather a high-level approximation of overall model complexity. Performance is calculated as the execution time of a configuration normalized against the execution time of a simple object-oriented instruction set simulator implemented in Python and executed using PyPy. This instruction set simulator is given an LOD score of 1 since it consists of only a single FL component, and it is plotted at coordinate coordinate (1,1) in Figure 3.15.

A general downward trend is observed in relative simulation performance as LOD increases. This is due to the greater computational effort required to simulate increasingly detailed models, resulting in a corresponding increase in execution time. In particular, a significant drop in performance can be seen between the simple instruction set simulator (LOD = 1) and the $\langle$FL,FL,FL$\rangle$

configuration (LOD = 3). This gap demonstrates the costs associated with modular modeling of components, structural composition, and communication overheads incurred versus a monolithic implementation with a tightly integrated memory and accelerator implementation. Occasionally, a model with a high LOD will take less execution time than a model with low LOD. For CPython data points this is largely due to more detailed models taking advantage of pipelining or parallelism to reduce target execution cycles. For example, the FL model of the accelerator does not pipeline memory operations and therefore executes many more target cycles than the CL implementation. For SimJIT+PyPy data points the effectiveness of each specialization strategy and the complexity of each component being specialized plays an equally significant role. FL components only benefit from the optimizations provided by PyPy and in some cases may perform worse than CL or RTL models which benefit from both SimJIT and PyPy, despite their greater LOD. Section 3.6.3 explores the performance characteristics of each specialization strategy in more detail.

Comparing the SimJIT+PyPy and CPython data points we can see that just-in-time specialization is able to significantly improve the execution time of each configuration, resulting in a vertical shift that makes even the most detailed models competitive with the CPython versions of simple models. Even better results could be expected if SimJIT-CL optimizations were applied to CL processor and CL accelerator models as well. Of particular interest is the ⟨RTL,RTL,RTL⟩ configuration (LOD = 9) which demonstrates better simulation performance than many less detailed configurations. This is because all subcomponents of the model can be optimized together as a monolithic unit, further reducing the overhead of Python wrapping. More generally, Figure 3.15 demonstrates the impact of two distinct approaches to improving PyMTL performance: (1) improvements that can be obtained *automatically* through specialization using SimJIT+PyPy, and (2) improvements that can be obtained *manually* by tailoring simulation detail via multi-level modeling.

3.6.3   SimJIT Performance: Mesh Network

We use the mesh network discussed in Section 3.5.2 to explore in greater detail the performance impact and overheads associated with SimJIT and PyPy. A network makes a good model for this purpose, since it allows us to flexibly configure size, injection rate, and simulation time to examine SimJIT's performance on models of varying complexity and under various loads. Figure 3.16 shows the impact of just-in-time specialization on 64-node FL, CL, and RTL mesh networks near

(a) FL Network

(b) CL Network

(c) RTL Network
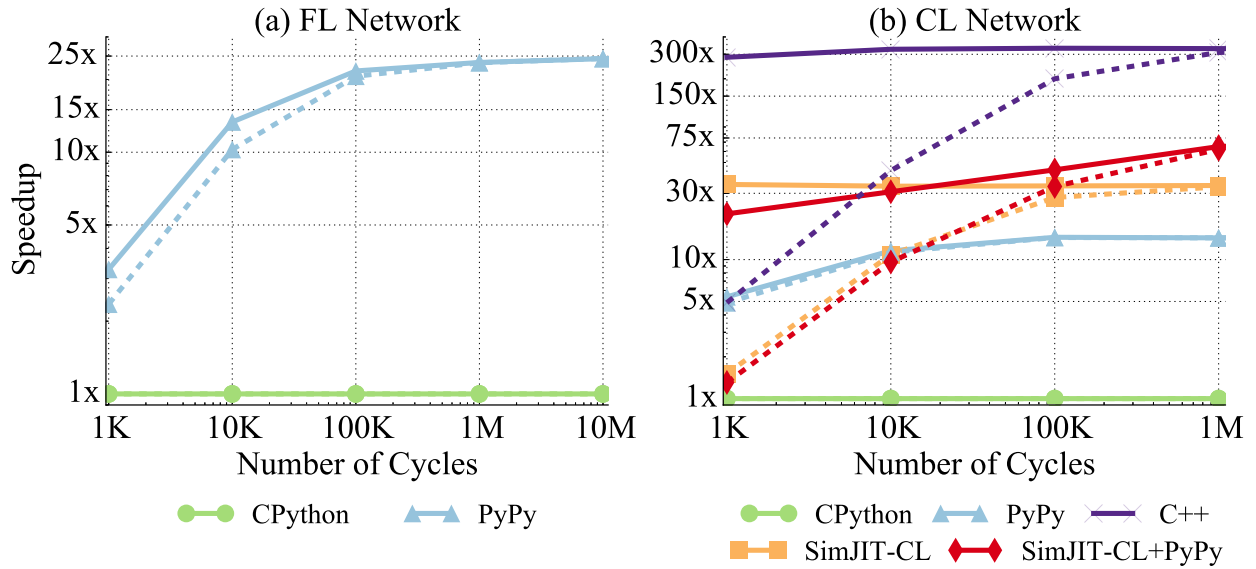
Figure 3.16: SimJIT Mesh Network Performance – Simulation of 64-node FL, CL, and RTL mesh network models operating near saturation. Dotted lines indicate total simulation-time speedup including all compilation and specialization overheads. Solid lines indicate speedup ignoring all overheads shown in Figure 3.18. For CL and RTL models, the solid lines closely approximate the speedup seen with caching enabled. No SimJIT optimization exists for FL models, but PyPy is able to provide good speedups. SimJIT+PyPy brings CL/RTL execution time within 4×/6× of C++/Verilator simulation, respectively.
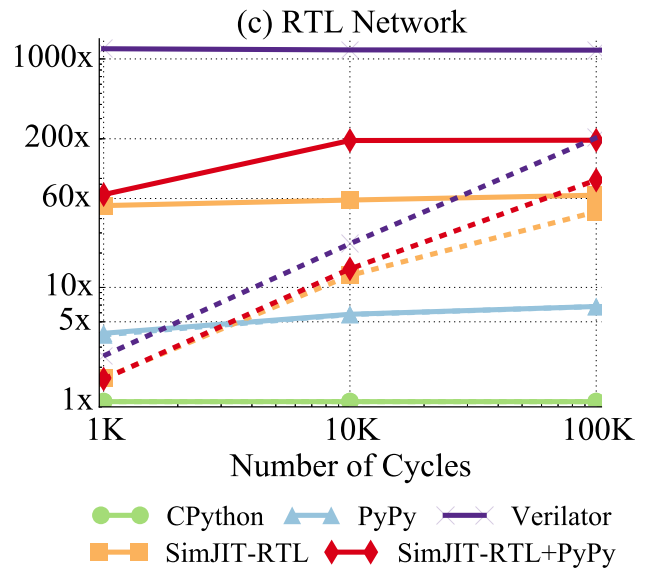
saturation. All results are normalized to the performance of CPython. Dotted lines show speedup of *total* simulation time while solid lines indicate speedup after *subtracting the simulation overheads shown in Figure 3.18*. These overheads are discussed in detail later in this section. Note that the dotted lines in Figure 3.16 are the real speedup observed when running a single experiment, while the solid line is an approximation of the speedup observed when caching is available. Our SimJIT-RTL caching implementation is able to remove the *compilation* and *verilation* overheads (shown in Figure 3.18) so the solid line closely approximates the speedups seen when doing multiple simulations of the same model instance.

The FL network plot in Figure 3.16(a) compares only PyPy versus CPython execution since no embedded-specializer exists for FL models. PyPy demonstrates a speedup between 2–25× depending on the length of the simulation. The bend in the solid line represents the warm-up time associated with PyPy's tracing JIT. After 10M target cycles the JIT has completely warmed-up and almost entirely amortizes all JIT overheads. The only overhead included in the dotted line is elaboration, which has a performance impact of less than a second.

The CL network plot in Figure 3.16(b) compares PyPy, SimJIT-CL, SimJIT-CL+PyPy, and a hand-coded C++ implementation against CPython. The C++ implementation is implemented using an in-house concurrent-structural modeling framework in the same spirit as Liberty [VVP⁺06] and Cascade [GTBS13]. It is designed to have cycle-exact simulation behavior with respect to the PyMTL model and is driven with an identical traffic pattern. The pure C++ implementation sees a speedup over CPython of up to 300× for a 10M-cycle simulation, but incurs a significant overhead from compilation time (dotted line). While this overhead is less important when model design has completed and long simulations are being performed for evaluation, this time significantly impacts the code-test-debug loop of the programmer, particularly when changing a module that forces a rebuild of many dependent components. An interpreted design language provides a significant productivity boost in this respect as simulations of less than 1K target cycles (often used for debugging) offer quicker turn around than a compiled language. For long runs of 10M target cycles, PyPy is able to provide a 12× speedup over CPython, SimJIT a speedup of 30×, and the combination of SimJIT and PyPy a speedup of 75×; this brings us within 4× of hand-coded C++.

The RTL network plot in Figure 3.16(c) compares PyPy, SimJIT-RTL, SimJIT-RTL+PyPy, and a hand-coded Verilog implementation against CPython. For the Verilog network we use Verilator to generate a C++ simulator, manually write a C++ test harness, and compile them together to create a simulator binary. Again, the Verilog implementation has been verified to be cycle-exact with our PyMTL implementation and is driven using an identical traffic pattern. Due to the detailed nature of RTL simulation, Python sees an even greater performance degradation when compared to C++. For the longest running configuration of 10M target cycles, C++ observes a 1200× speedup over CPython. While this performance difference makes Python a non-starter for long running simulations, achieving this performance comes at a significant compilation overhead: compiling Verilator-generated C++ for the 64-node mesh network takes over 5 minutes using the relatively fast -O1 optimization level of GCC. PyPy has trouble providing significant speedups over more

54
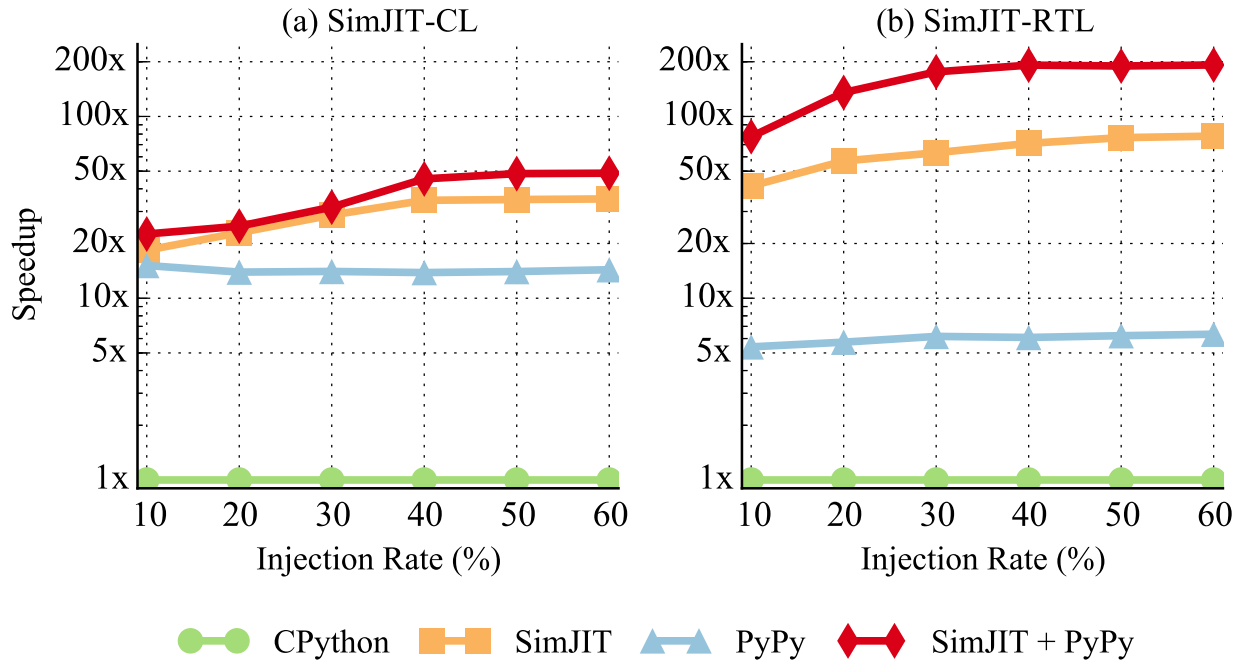
Figure 3.17: SimJIT Performance vs. Load – Impact of injection rate on a 64-node network simulation executing for 100K cycles. Heavier load results in longer execution times, enabling overheads to be amortized more rapidly for a given number of simulated cycles as more time is spent in optimized code.

complicated designs, and in this case only achieves a $6\times$ improvement over CPython. SimJIT-RTL provides a $63\times$ speedup and combining SimJIT-RTL with PyPy provides a speedup of $200\times$, bringing us within $6\times$ of verilated hand-coded Verilog.

To explore how simulator activity impacts our SimJIT speedups, we vary the injection rate of the 64-node mesh network simulations for both the CL and RTL models (see Figure 3.17). In comparison to CPython, PyPy performance is relatively consistent across loads, while SimJIT-CL and SimJIT-RTL see increased performance under greater load. SimJIT speedup ranges between 23–$49\times$ for SimJIT-CL+PyPy and 77–$192\times$ for SimJIT-RTL+PyPy. The curves of both plots begin to flatten out at the network's saturation point near an injection rate of 30%. This is due to the increased amount of execution time being spent inside the network model during each simulation tick meaning more time is spent in optimized C++ code for the SimJIT configurations.

The overheads incurred by SimJIT-RTL and SimJIT-CL increase with larger model sizes due to the increased quantity of code that must be generated and compiled. Figure 3.18 shows these overheads for 4×4 and 8×8 mesh networks. These overheads are relatively modest for SimJIT-RTL at under 5 and 20 seconds for the 16- and 64-node meshes, respectively. The use of PyPy
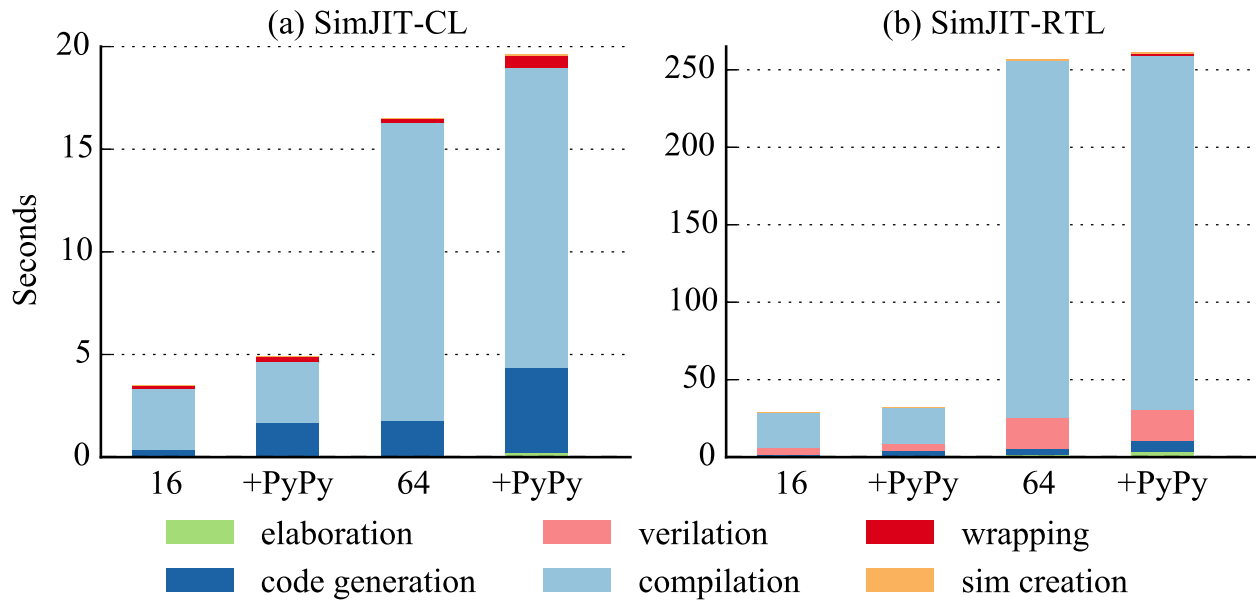
Figure 3.18: SimJIT Overheads – Elaboration, code generation, verilation, compilation, Python wrapping (wrapping), and sim creation all contribute overhead to run-time construction of specializers. Compile time has the largest impact for both SimJIT-RTL and SimJIT-CL. Verilation, which is not present in SimJIT-CL, has a significant impact for SimJIT-RTL, especially for larger models.

slightly increases the overhead of SimJIT. This is because SimJIT's elaboration, code generation, wrapping, and simulator creation phases are all too short to amortize PyPy's tracing JIT overhead. However, this slowdown is negligible compared to the significant speedups PyPy provides during simulation. SimJIT-RTL has an additional verilation phase, as well as significantly higher compilation times: 22 seconds for a 16-node mesh and 230 seconds for a 64-node mesh. Fortunately, the overheads for verilation, compilation, and wrapping can be converted into a one-time cost using SimJIT-RTL's simple caching scheme.

3.7 Related Work

A number of previous projects have proposed using Python for hardware design. Stratus, PHDL, and PyHDL generate HDL from parameterized structural descriptions in Python by using provided library blocks, but do not provide simulation capabilities or support for FL or CL modeling [BDM+07, Mas07, HMLT03]. MyHDL uses Python as a hardware description language that can be simulated in a Python interpreter or translated to Verilog and VHDL [Dec04, VJB+11]. SysPy is a tool intended to aid processor-centric SoC designs targeting FPGAs that integrates with existing IP and user-provided C source source [LM10]. PDSDL, enables behavioral and structural description of RTL models that can be simulated within a Python-based kernel, as well as translated into HDL. PDSDL was used in the construction of Trilobyte, a framework for refining behavioral processor descriptions into HDL [ZTC08, ZHCT09]. Other than PDSDL, the above frameworks focus primarily on structural or RTL hardware descriptions and do not address higher level modeling. In addition, none attempt to address the performance limitations inherent to using Python for simulation.

Hardware generation languages help address the need for rapid design-space exploration and collection of area, energy, and timing metrics by making RTL design more productive. Genesis2 combined SystemVerilog with Perl scripts to create highly parameterizable hardware designs for the creation of chip generators [SWD+12, SAW+10]. Chisel is an HDL implemented as an EDSL within Scala. Hardware descriptions in Chisel are translated into to either Verilog HDL or C++ simulations. There is no Scala simulation of hardware descriptions [BVR+12]. BlueSpec is an HGL built on SystemVerilog that describes hardware using guarded atomic actions [Nik04, HA03].

A number of other simulation frameworks have applied a concurrent-structural modeling approach to cycle-level simulation. The Liberty Simulation Environment argued that concurrent-structural modeling greatly improved understanding and reuse of components, but provided no HDL integration or generation [VVP+02, VVA04, VVP+06]. Cascade is a concurrent-structural simulation framework used in the design and verification of the Anton supercomputers. Cascade provides tight integration with an RTL flow by enabling embedding of Cascade models within Verilog test harnesses as well as Verilog components within Cascade models [GTBS13]. SystemC also leverages a concurrent-structural design methodology that was originally intended to provide an integrated framework for multiple levels of modeling and refinement to implementation, including

a synthesizable language subset. Unfortunately, most of these thrusts did not see wide adoption and SystemC is currently used primarily for the purposes of virtual system prototyping and high level synthesis [Pan01, sys14].

While significant prior work has explored generation of optimized simulators including work by Penry et al. [PA03, PFH+06, Pen06], to our knowledge there has been no previous work on using just-in-time compilation to speed up CL and RTL simulations using dynamically-typed languages. SEJITS proposed just-in-time specialization of high-level algorithm descriptions written in dynamic languages into optimized, platform-specific multicore or CUDA source [CKL+09]. JIT techniques have also been previously leveraged to accelerate instruction set simulators (ISS) [May87, CK94, WR96, MZ04, TJ07, WGFT13]. The GEZEL environment combines a custom interpreted DSL for coprocessor design with existing ISS, supporting both translation into synthesizable VHDL and simulation-time conversion into C++ [SCV06]. Unlike PyMTL, GEZEL is not a general-purpose language and only supports C++ translation of RTL models; PyMTL supports JIT specialization of CL and RTL models.

3.8 Conclusion

This chapter has presented PyMTL, a unified, vertically integrated framework for FL, CL, and RTL modeling. Small case studies were used to illustrate how PyMTL can close the computer architecture methodology gap by enabling productive construction of composable FL, CL, and RTL models using concurrent-structural and latency-insensitive design. While these small examples demonstrated some of the power of PyMTL, PyMTL is just a first step towards enabling rapid design-space exploration and construction of flexible hardware templates to amortize design effort. In addition, a hybrid approach to just-in-time optimization was proposed to close the performance gap introduced by using Python for hardware modeling. SimJIT, a custom JIT specializer for CL and RTL models, was combined with the PyPy meta-tracing JIT interpreter to bring PyMTL simulation of a mesh network within 4×–6× of optimized C++ code.

A key contribution of this work is the idea that combining embedded-DSLs with JIT specializers can be used to construct a productive, vertically integrated hardware design methodology. The ultimate goal of this approach is to achieve the benefits of both efficiency-level language productivity for hardware design and productivity-level language performance for efficient simulation.

While prior work on SEJITS has proposed similar techniques for improving the productivity of working with domain-specific algorithms, such as stencil and graphs computations, PyMTL has demonstrated this approach can be applied in a much more general manner for designing hardware at multiple levels of abstraction. The PyMTL framework has also shown that such an approach is a promising technique for the construction of future hardware design tools.

CHAPTER 4
PYDGIN: FAST INSTRUCTION SET SIMULATORS FROM SIMPLE SPECIFICATIONS

In the previous chapter, the use of SimJIT was able to significantly improve the simulation performance of PyMTL cycle-level (CL) and register-transfer-level (RTL) models. Constructing a similar SimJIT for functional-level (FL) models is a much more challenging problem because these models generally contain arbitrary Python and frequently take advantage of powerful language features needed to quickly and concisely implement complex algorithms. For many FL models the PyPy interpreter can provide sufficient speedups due to its general-purpose tracing-JIT; for example, PyPy was able to provide up to a $25\times$ improvement in simulation performance for the FL network in Figure 3.16. However, instruction set simulators (ISSs) are a particularly important class of FL model that have a critical need for high performance simulation. This performance is needed to enable the execution of large binary executables for application development, but PyPy alone cannot achieve the performance fidelity needed to implement a practicable ISS.

This chapter describes Pydgin: a framework that generates fast instruction set simulators with dynamic binary translation (DBT) from a simple, Python-based, embedded architectural description language (ADL). Background information on the RPython translation toolchain is provided; this toolchain is a framework for implementing dynamic language interpreters with meta-tracing JITs that is adapted by Pydgin for use in creating fast ISSs. Pydgin's Python-based, embedded-ADL is described and compared to ADLs used in other ISS-generation frameworks. ISS-specific JIT annotations added to Pydgin are described and their performance impact is analyzed. Finally, three ISSs constructed using Pydgin are benchmarked and evaluated.

4.1 Introduction

Recent challenges in CMOS technology scaling have motivated an increasingly fluid boundary between hardware and software. Examples include new instructions for managing fine-grain parallelism, new programmable data-parallel engines, programmable accelerators based on reconfigurable coarse-grain arrays, domain-specific co-processors, and a rising demand for application-specific instruction set processors (ASIPs). This trend towards heterogeneous hardware/software abstractions combined with complex design targets is placing increasing importance on highly productive and high-performance instruction set simulators (ISSs).

Unfortunately, meeting the multitude of design requirements for a modern ISS (observability, retargetability, extensibility, and support for self-modifying code) while also providing productivity and high performance has led to considerable ISS design complexity. Highly productive ISSs have adopted architecture description languages (ADLs) as a means to enable abstract specification of instruction semantics and simplify the addition of new instruction set features. The ADLs in these frameworks are domain specific languages constructed to be sufficiently expressive for describing traditional architectures, yet restrictive enough for efficient simulation (e.g., ArchC [RABA04, ARB$^+$05], LISA [vPM96, PHM00], LIS [Pen11], MADL [QRM04, QM05], SimIt-ARM ADL [DQ06, QDZ06]). In addition, high-performance ISSs use dynamic binary translation (DBT) to discover frequently executed blocks of target instructions and convert these blocks into optimized sequences of host instructions. DBT-ISSs often require a deep understanding of the target instruction set in order to enable fast and efficient translation. However, promising recent work has demonstrated sophisticated frameworks that can automatically generate DBT-ISSs from ADLs [PC11, WGFT13, QM03b].

Meanwhile, designers working on interpreters for general-purpose dynamic programming languages (e.g., Javascript, Python, Ruby, Lua, Scheme) face similar challenges balancing productivity of interpreter developers with performance of the interpreter itself. The highest performance interpreters use just-in-time (JIT) trace- or method-based compilation techniques. As the sophistication of these techniques have grown so has the complexity of interpreter codebases. For example, the WebKit Javascript engine currently consists of four distinct tiers of JIT compilers, each designed to provide greater amounts of optimization for frequently visited code regions [Piz14]. In light of these challenges, one promising approach introduced by the PyPy project uses *meta-tracing* to greatly simplify the design of high-performance interpreters for dynamic languages. PyPy's meta-tracing toolchain takes traditional interpreters implemented in RPython, a restricted subset of Python, and automatically translates them into optimized, tracing-JIT compilers [BCF$^+$11, BCFR09, AACM07, pyp11, Pet08]. The RPython translation toolchain has been previously used to rapidly develop high-performance JIT-enabled interpreters for a variety of different languages [BLS10, BKL$^+$08, BPSTH14, Tra05, Tho13, top15, hip15]. A key observation is that similarities between ISSs and interpreters for dynamic programming languages suggest that the RPython translation toolchain might enable similar productivity and performance benefits when applied to instruction set simulator design.

This chapter introduces Pydgin[1], a new approach to ISS design that combines an embedded-ADL with automatically-generated meta-tracing JIT interpreters to close the productivity-performance gap for future ISA design. The Pydgin library provides an embedded-ADL within RPython for succinctly describing instruction semantics, and also provides a modular instruction set interpreter that leverages these user-defined instruction definitions. In addition to mapping closely to the pseudocode-like syntax of ISA manuals, Pydgin instruction descriptions are fully executable within the Python interpreter for rapid code-test-debug during ISA development. The RPython translation toolchain is adapted to take Pydgin ADL descriptions as input, and automatically convert them into high-performance DBT-ISSs. Building the Pydgin framework required approximately three person-months worth of work, but implementing two different instruction sets (a simple MIPS-based instruction set and a more sophisticated ARMv5 instruction set) took just a few weeks and resulted in ISSs capable of executing many of the SPEC CINT2006 benchmarks at hundreds of millions of instructions per second. More recently, a third ISS implementation for the 64-bit RISC-V ISA was also created using Pydgin. This ISS implemented the entirety of the M, A, F, and D extensions of the RISC-V ISA and yet was completed in under two weeks, further affirming the productivity of constructing instruction set simulators in Pydgin.

4.2 The RPython Translation Toolchain

The increase in popularity of dynamic programming languages has resulted in a significant interest in high-performance interpreter design. Perhaps the most notable examples include the numerous JIT-optimizing JavaScript interpreters present in modern browsers today. Another example is PyPy, a JIT-optimizing interpreter for the Python programming language. PyPy uses JIT compilation to improve the performance of hot loops, often resulting in considerable speedups over the reference Python interpreter, CPython. 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 RPython translation toolchain enables developers to describe an interpreter in a restricted subset of Python (called RPython) and then automatically translate this RPython interpreter implementation into

---

[1]Pydgin loosely stands for [Py]thon [D]SL for [G]enerating [In]struction set simulators and is pronounced the same as "pigeon". The name is inspired by the word "pidgin" which is a grammatically simplified form of language and captures the intent of the Pydgin embedded-ADL.

a C executable. With the addition of a few basic annotations, the RPython translation toolchain can also automatically insert a tracing-JIT compiler into the generated C-based interpreter. In this section, we briefly describe the RPython translation toolchain, which we leverage as the foundation for the Pydgin framework. More detailed information about RPython and the PyPy project in general can be found in [Bol12, BCF$^+$11, BCFR09, AACM07, pyp11, Pet08].

Python is a dynamically typed language with typed objects but untyped variable names. RPython is a carefully chosen subset of Python that enables static type inference such that the type of both objects and variable names can be determined at translation time. Even though RPython sacrifices some of Python's dynamic features (e.g., duck typing, monkey patching) it still maintains many of the features that make Python productive (e.g., simple syntax, automatic memory management, large standard library). In addition, RPython supports powerful meta-programming allowing full-featured Python code to be used to generate RPython code at translation time.

Figure 4.1 shows a simple bytecode interpreter and illustrates how interpreters written in RPython can be significantly simpler than a comparable interpreter written in C (example adapted from [BCFR09]). The example is valid RPython because the type of all variables can be determined at translation time (e.g., `regs`, `acc`, and `pc` are always of type `int`; bytecode is always of type `str`). Figure 4.2(a) shows the RPython translation toolchain. The *elaboration phase* can use full-featured Python code to generate RPython source as long as the interpreter loop only contains valid RPython prior to starting the next phase of translation. The *type inference phase* uses various algorithms to determine high-level type information about each variable (e.g., integers, real numbers, user-defined types) before lowering this type information into an annotated intermediate representation (IR) with specific C datatypes (e.g., `int`, `long`, `double`, `struct`). The *back-end optimization phase* leverages standard optimization passes to inline functions, remove unnecessary dynamic memory allocation, implement exceptions efficiently, and manage garbage collection. The *code generation phase* translates the optimized IR into C source code, before the *compilation phase* generates the C-based interpreter.

The RPython translation toolchain also includes support for automatically generating a tracing JIT compiler to complement the generated C-based interpreter. To achieve this, the RPython toolchain uses a novel *meta-tracing* approach where the JIT compiler does not directly trace the bytecode but instead traces the *interpreter* interpreting the bytecode. While this may initially seem counter-intuitive, meta-tracing JIT compilers are the key to improving productivity and perfor-

63

```
jd = JitDriver( greens = ['bytecode', 'pc'],
                reds   = ['regs', 'acc'] )

def interpreter( bytecode ):
  regs = [0]*256 # vm state: 256 registers
  acc  = 0        # vm state: accumulator
  pc   = 0        # vm state: program counter

  while True:
    jd.jit_merge_point( bytecode, pc, regs, acc )
    opcode = ord(bytecode[pc])
    pc += 1

    if opcode == JUMP_IF_ACC:
      target = ord(bytecode[pc])
      pc += 1
      if acc:
        if target < pc:
          jd.can_enter_jit( bytecode, pc, regs, acc )
        pc = target

    elif opcode == MOV_ACC_TO_REG:
      rs = ord(bytecode[pc])
      regs[rs] = acc
      pc += 1

    # ... handle remaining opcodes ...
```

Figure 4.1: Simple Bytecode Interpreter Written in RPython – `bytecode` is string of bytes encoding instructions that operate on 256 registers and an accumulator. RPython enables succinct interpreter descriptions that can still be automatically translated into C code. Basic annotations (shown in blue) enable automatically generating a meta-tracing JIT compiler. Adapted from [BCFR09].

mance. This approach decouples the design of the interpreter, which can be written in a high-level dynamic language such as RPython, from the complexity involved in implementing a tracing JIT compiler for that interpreter. A direct consequence of this separation of concerns is that interpreters for different languages can all leverage the exact same JIT compilation framework as long as these interpreters make careful use of *meta-tracing annotations*.

Figure 4.1 highlights the most basic meta-tracing annotations required to automatically generate reasonable JIT compilers. The `JitDriver` object instantiated on lines 1–2 informs the JIT of which variables identify the interpreter's position within the target application's bytecode (greens), and which variables are not part of the position key (reds). The `can_enter_jit` annotation on line 19 tells the JIT where an *application-level loop* (i.e., a loop in the actual bytecode application) begins; it is used to indicate backwards-branch bytecodes. The `jit_merge_point`
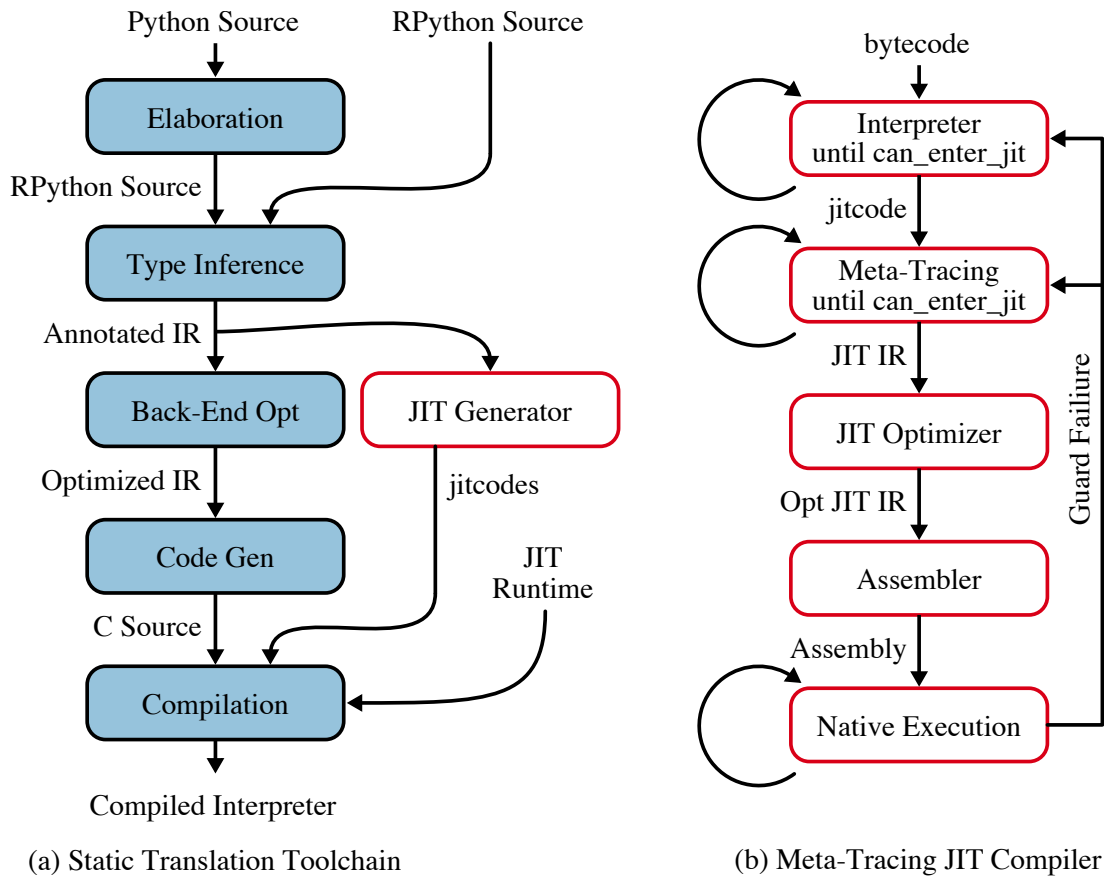
Figure 4.2: RPython Translation Toolchain – (a) the static translation toolchain converts an RPython interpreter into C code along with a generated JIT compiler; (b) the meta-tracing JIT compiler traces the interpreter (not the application) to eventually generate optimized assembly for native execution.

annotation on line 10 tells the JIT where it is safe to move from the JIT back into the interpreter; it is used to identify the top of the interpreter's dispatch loop. As shown in Figure 4.2(a), the JIT generator replaces the `can_enter_jit` hints with calls into the JIT runtime and then serializes the annotated IR for all code regions between the meta-tracing annotations. These serialized IR representations are called "jitcodes" and are integrated along with the JIT runtime into the C-based interpreter. Figure 4.2(b) illustrates how the meta-tracing JIT compiler operates at runtime. When the C-based interpreter reaches a `can_enter_jit` hint, it begins using the corresponding jitcode to build a meta-trace of the interpreter interpreting the bytecode application. When the same `can_enter_jit` hint is reached again, the JIT increments an internal per-loop counter. Once this counter exceeds a threshold, the collected trace is handed off to a JIT optimizer and assembler before initiating native execution. The meta-traces include guards that ensure the dynamic
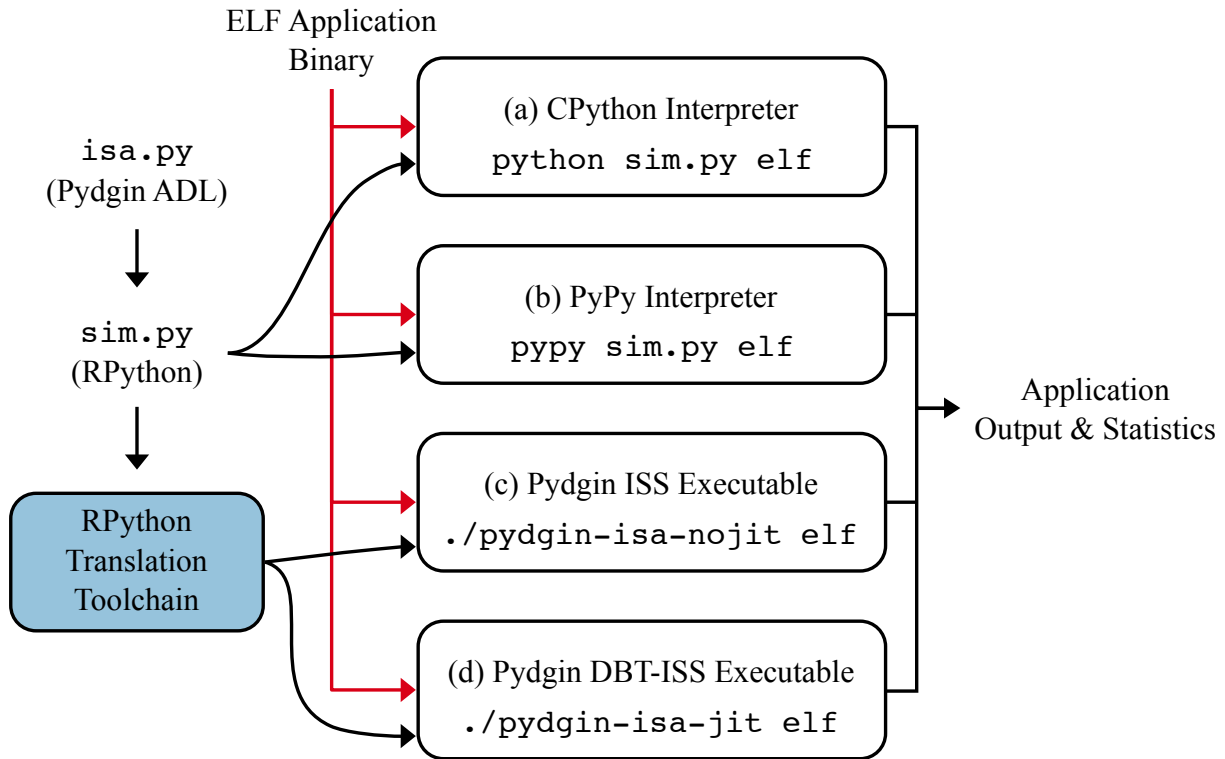
Figure 4.3: Pydgin Simulation – Pydgin ISA descriptions are imported by the Pydgin simulation driver which defines the top-level interpreter loop. The resulting Pydgin ISS can be executed directly using (a) the reference CPython interpreter or (b) the higher-performance PyPy JIT-optimizing interpreter. Alternatively, the interpreter loop can be passed to the RPython translation toolchain to generate a C-based executable implementing (c) an interpretive ISS or (d) a DBT-ISS.

conditions under which the meta-trace was optimized still hold (e.g., the types of application-level variables remain constant). If at any time a guard fails or if the optimized loop is finished, then the JIT returns control back to the C-based interpreter at a `jit_merge_point`.

Figure 4.3 illustrates how the RPython translation toolchain is leveraged by the Pydgin framework. Once an ISA has been specified using the Pydgin embedded-ADL (described in Section 4.3) it is combined with the Pydgin simulation driver, which provides a modular, pre-defined interpreter implementation, to create an executable Pydgin instruction set simulator. Each Pydgin ISS is valid RPython that can be executed in a number of ways. The most straightforward execution is direct interpretation using CPython or PyPy. Although interpreted execution provides poor simulation performance, it serves as a particularly useful debugging platform during early stages of ISA development and testing. Alternatively, the Pydgin ISS can be passed as input to the RPython

```
1  class State( object ):
2    _virtualizable_ = ['pc', 'ncycles']
3    def __init__( self, memory, debug, reset_addr=0x400 ):
4      self.pc      = reset_addr
5      self.rf      = ArmRegisterFile( self, num_regs=16 )
6      self.mem     = memory
7
8      self.rf[ 15 ]  = reset_addr
9
10     # current program status register (CPSR)
11     self.N    = 0b0      # Negative condition
12     self.Z    = 0b0      # Zero condition
13     self.C    = 0b0      # Carry condition
14     self.V    = 0b0      # Overflow condition
15
16     # simulator/stats info, not architecturally visible
17     self.status  = 0
18     self.ncycles = 0
19
20   def fetch_pc( self ):
21     return self.pc
```

Figure 4.4: Simplified ARMv5 Architectural State Description

translation toolchain in order to generate a compiled executable implementing either an interpretive ISS or a high-performance DBT-ISS (described in Section 4.4).

4.3   The Pydgin Embedded-ADL

To evaluate the capabilities of the Pydgin framework, Pydgin was used to implement instruction set simulators for three ISAs: a simplified version of MIPS32 called SMIPS, a subset of the ARMv5 ISA, and the 64-bit RISC-V ISA. This process involves using the Pydgin embedded-ADL to describe the architectural state, instruction encoding, and instruction semantics of each ISA. No special parser is needed to generate simulators from Pydgin ISA definitions, and in fact these definitions can be executed directly using a standard Python interpreter. In this section, we describe the various components of the Pydgin embedded-ADL using the ARMv5 ISA as an example.

4.3.1   Architectural State

Architectural state in Pydgin is implemented using Python classes. Figure 4.4 shows a simplified version of this state for the ARMv5 ISA. Library components provided by the Pydgin

```
1  encodings = [
2    ['nop',      '00000000000000000000000000000000'],
3    ['mul',      'xxxx0000000xxxxxxxxxxxxx1001xxxx'],
4    ['umull',    'xxxx0000100xxxxxxxxxxxxx1001xxxx'],
5    ['adc',      'xxxx00x0101xxxxxxxxxxxxxxxxxxxxxx'],
6    ['add',      'xxxx00x0100xxxxxxxxxxxxxxxxxxxxxx'],
7    ['and',      'xxxx00x0000xxxxxxxxxxxxxxxxxxxxxx'],
8    ['b',        'xxxx1010xxxxxxxxxxxxxxxxxxxxxxxxx'],
9    ...
10   ['teq',      'xxxx00x10011xxxxxxxxxxxxxxxxxxxxx'],
11   ['tst',      'xxxx00x10001xxxxxxxxxxxxxxxxxxxxx'],
12 ]
```

Figure 4.5: Partial ARMv5 Instruction Encoding Table

embedded-ADL such as `RegisterFile` and `Memory` classes can be used as provided or subclassed to suit the specific needs of a particular architecture. For example, the `ArmRegisterFile` on line 5 subclasses the `RegisterFile` component (not shown) and specializes it for the unique idiosyncrasies of the ARM architecture: within instruction semantic definitions register 15 must update the current `PC` when written but return `PC+8` when read. The `fetch_pc` accessor on line 20 is used to retrieve the current instruction address, which is needed for both instruction fetch and incrementing the `PC` in instruction semantic definitions (discussed in Section 4.3.3). Users may also implement their own data structures, however, these data structures must conform to the restrictions imposed by the RPython translation toolchain. The class member `_virtualizable_` is an optional JIT annotation used by the RPython translation toolchain. We discuss this and other advanced JIT annotations in more detail in Section 4.4.

4.3.2 Instruction Encoding

Pydgin maintains encodings of all instructions in a centralized data structure for easy maintenance and quick lookup. A partial encoding table for the ARMv5 instruction set can be seen in Figure 4.5. While some ADLs keep this encoding information associated with the each instruction's semantic definition (e.g., SimIt-ARM's definitions in Figure 4.8), we have found that this distributed organization makes it more difficult to quickly assess available encodings for introducing ISA extensions. However, ISA designers preferring this distributed organization can easily implement it using Python decorators to annotate instruction definitions with their encoding. This illustrates the power of using an embedded-ADL where arbitrary Python code can be used for

68

metaprogramming. Encodings and instruction names provided in the instruction encoding table are used by Pydgin to automatically generate decoders for the simulator. Unlike some ADLs, Pydgin does not require the user to explicitly specify instruction types or mask bits for field matching because Pydgin can automatically infer field locations from the encoding table.

4.3.3 Instruction Semantics

Pydgin instruction semantic definitions are implemented using normal Python functions with the special signature `execute_<inst_name>(s,inst)`. The function parameters `s` and `inst` refer to the *architectural state* and the *instruction bits*, respectively. An example of a Pydgin instruction definition is shown for the ARMv5 `ADD` instruction in Figure 4.6. Pydgin allows users to create helper functions that refactor complex operations common across many instruction definitions. For example, `condition_passed` on line 2 performs predication checks, while `shifter_operand` on line 4 encapsulates ARMv5's complex rules for computing the secondary operand `b` and computes a special carry out condition needed by some instructions (stored in `cout`). This encapsulation provides the secondary benefit of helping Pydgin definitions better match the instruction semantics described in ISA manuals. Note that the Pydgin definition for `ADD` is a fairly close match to the instruction specification pseudocode provided in the official ARM ISA manual, shown in Figure 4.7.

Figure 4.8 shows another description of the `ADD` instruction in the SimIt-ARM ADL, a custom, lightweight ADL used by the open-source SimIt-ARM simulator to generate both interpretive and DBT ISSs [DQ06]. In comparison to the SimIt-ARM ADL, Pydgin is slightly less concise as a consequence of using an embedded-ADL rather than implementing a custom parser. The SimIt-ARM description implements `ADD` as four separate instructions in order to account for the `S` and `I` instruction bits. These bits determine whether condition flags are updated and if a rotate immediate addressing mode should be used, respectively. This multi-instruction approach is presumably done for performance reasons as splitting the `ADD` definition into separate instructions results in simpler decode and less branching behavior during simulation. However, this approach incurs additional overhead in terms of clarity and maintainability. Pydgin largely avoids the need for these optimizations thanks to its meta-tracing JIT compiler that can effectively optimize away branching behavior for hot paths. This works particularly well for decoding instruction fields such as the ARM conditional bits and the `S` and `I` flags: for non-self modifying code an instruction at a particular PC will

```
1  def execute_add( s, inst ):
2    if condition_passed( s, inst.cond() ):
3      a         = s.rf[ inst.rn() ]
4      b, cout   = shifter_operand( s, inst )
5
6      result    = a + b
7      s.rf[ inst.rd() ] = trim_32( result )
8
9      if inst.S():
10       if inst.rd() == 15:
11         raise Exception('Writing SPSR not implemented!')
12       s.N = (result >> 31)&1
13       s.Z = trim_32( result ) == 0
14       s.C = carry_from( result )
15       s.V = overflow_from_add( a, b, result )
16
17     if inst.rd() == 15:
18       return
19
20   s.rf[PC] = s.fetch_pc() + 4
```

Figure 4.6: ADD Instruction Semantics: Pydgin

```
1  if ConditionPassed(cond) then
2    Rd = Rn + shifter_operand
3    if S == 1 and Rd == R15 then
4      if CurrentModeHasSPSR() then CPSR = SPSR
5    else UNPREDICTABLE else if S == 1 then
6      N Flag = Rd[31]
7      Z Flag = if Rd == 0 then 1 else 0
8      C Flag = CarryFrom(Rn + shifter_operand)
9      V Flag = OverflowFrom(Rn + shifter_operand)
```

Figure 4.7: ADD Instruction Semantics: ARM ISA Manual

always have the same instruction bits, enabling the JIT to completely optimize away this complex decode overhead.

An ArchC description of the ADD instruction can be seen in Figure 4.9. Note that some debug code has been removed for the sake of brevity. ArchC is an open-source, SystemC-based ADL popular in system-on-chip toolflows [RABA04, ARB+05]. ArchC has considerably more syntactic overhead than both the SimIt-ARM ADL and Pydgin embedded-ADL. Much of this syntactic overhead is due to ArchC's C++-style description which requires explicit declaration of complex templated types. One significant advantage ArchC's C++-based syntax has over SimIt-ARM's ADL is that it is compatible with existing C++ development tools. Pydgin benefits from RPython's

```
1  op add(----00001000:rn:rd:shifts) {
2  execute="
3    WRITE_REG($rd$, READ_REG($rn$) + $shifts$);
4  "
5  }
6
7  op adds(----00001001:rn:rd:shifts) {
8  execute="
9    tmp32 = $shifts$; val32 = READ_REG($rn$);
10   rslt32 = val32 + tmp32;
11     if ($rd$==15) WRITE_CPSR(SPSR);
12     else ASGN_NZCV(rslt32, rslt32<val32,
13         (val32^tmp32^-1) & (val32^rslt32));
14   WRITE_REG($rd$, rslt32);
15 "
16 }
17
18 op addi(----00101000:rn:rd:rotate_imm32) {
19 execute="
20   WRITE_REG($rd$, READ_REG($rn$) + $rotate_imm32$);
21 "
22 }
23
24 op addis(----00101001:rn:rd:rotate_imm32) {
25 execute="
26   tmp32 = $rotate_imm32$; val32 = READ_REG($rn$);
27   rslt32 = val32 + tmp32;
28   if ($rd$==15) WRITE_CPSR(SPSR);
29   else ASGN_NZCV(rslt32, rslt32<val32,
30       (val32^tmp32^-1) & (val32^rslt32));
31   WRITE_REG($rd$, rslt32);
32 "
33 }
```

Figure 4.8: ADD Instruction Semantics: SimIt-ARM

dynamic typing to produce a comparatively cleaner syntax while also providing compatibility with Python development tools.

4.3.4 Benefits of an Embedded-ADL

While the dynamic nature of Python enables Pydgin to provide relatively concise, pseudo-code-like syntax for describing instructions, it could be made even more concise by implementing a DSL which uses a custom parser. From our experience, embedded-ADLs provide a number of advantages over a custom DSL approach: increased language familiarity and a gentler learning curve for new users; access to better development tools and debugging facilities; and easier main-

71

```
1  inline void ADD(arm_isa* ref, int rd, int rn, bool s,
2      ac_regbank<31, arm_parms::ac_word,
3                     arm_parms::ac_Dword>& RB,
4      ac_reg<unsigned>& ac_pc) {
5
6    arm_isa::reg_t RD2, RN2;
7    arm_isa::r64bit_t soma;
8
9    RN2.entire = RB_read(rn);
10   if(rn == PC) RN2.entire += 4;
11   soma.hilo = (uint64_t)(uint32_t)RN2.entire +
12       (uint64_t)(uint32_t)ref->dpi_shiftop.entire;
13   RD2.entire = soma.reg[0];
14   RB_write(rd, RD2.entire);
15   if ((s == 1)&&(rd == PC)) {
16     #ifndef FORGIVE_UNPREDICTABLE
17     ...
18     ref->SPSRtoCPSR();
19     #endif
20   } else {
21     if (s == 1) {
22       ref->flags.N = getBit(RD2.entire,31);
23       ref->flags.Z = ((RD2.entire==0) ? true : false);
24       ref->flags.C = ((soma.reg[1]!=0) ? true : false);
25       ref->flags.V = (((getBit(RN2.entire,31)
26           && getBit(ref->dpi_shiftop.entire,31)
27           && (!getBit(RD2.entire,31)))
28           || ((!getBit(RN2.entire,31))
29           && (!getBit(ref->dpi_shiftop.entire,31))
30           && getBit(RD2.entire,31))) ? true : false);
31     }
32   }
33   ac_pc = RB_read(PC);
34 }
```

Figure 4.9: ADD Instruction Semantics: ArchC

tenance and extension by avoiding a custom parser. Additionally, we have found that the ability
to directly execute Pydgin ADL descriptions in a standard Python interpreter such as CPython or
PyPy significantly helps debugging and testing during initial ISA exploration.

4.4   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
generated by the RPython translation toolchain using only the basic JIT annotations shown in

Figure 4.10 provides good but not exceptional performance. 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.

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$^+$11]. 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 4.10 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 4.11 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 4.12 and 4.13 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 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 4.11.

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 likely constant, then the JIT can use constant

```
1   jd = JitDriver( greens = ['pc'], reds = ['state'],
2                   virtualizables = ['state'] )
3
4   class State( object ):
5     _virtualizable_ = ['pc', 'ncycles']
6     def __init__( self, memory, reset_addr=0x400 ):
7       self.pc      = reset_addr
8       self.ncycles = 0
9       # ... more architectural state ...
10
11  class Memory( object ):
12    def __init__( self, size=2**10 ):
13      self.size  = size << 2
14      self.data  = [0] * self.size
15
16    def read( self, start_addr, num_bytes ):
17      word = start_addr >> 2
18      byte = start_addr &  0b11
19      if   num_bytes == 4:
20        return self.data[ word ]
21      elif num_bytes == 2:
22        mask = 0xFFFF << (byte * 8)
23        return (self.data[ word ] & mask) >> (byte * 8)
24      # ... handle single byte read ...
25
26    @elidable
27    def iread( self, start_addr, num_bytes ):
28      return self.data[ start_addr >> 2 ]
29
30    # ... rest of memory methods ...
31
32  def run( state, max_insts=0 ):
33    s = state
34    while s.status == 0:
35      jd.jit_merge_point( s.fetch_pc(), max_insts, s )
36
37      pc  = hint( s.fetch_pc(), promote=True )
38      old = pc
39      mem = hint( s.mem, promote=True )
40
41      inst = mem.iread( pc, 4 )
42      exec_fun = decode( inst )
43      exec_fun( s, inst )
44
45      s.ncycles += 1
46
47      if s.fetch_pc() < old:
48        jd.can_enter_jit( s.fetch_pc(), max_insts, s )
49
```

Figure 4.10: Simplified Instruction Set Interpreter Written in RPython – Although only basic annotations (shown in blue) are required by the RPython translation toolchain to produce a JIT, more advanced annotations (shown in red) are needed to successfully generate efficient DBT-ISSs.
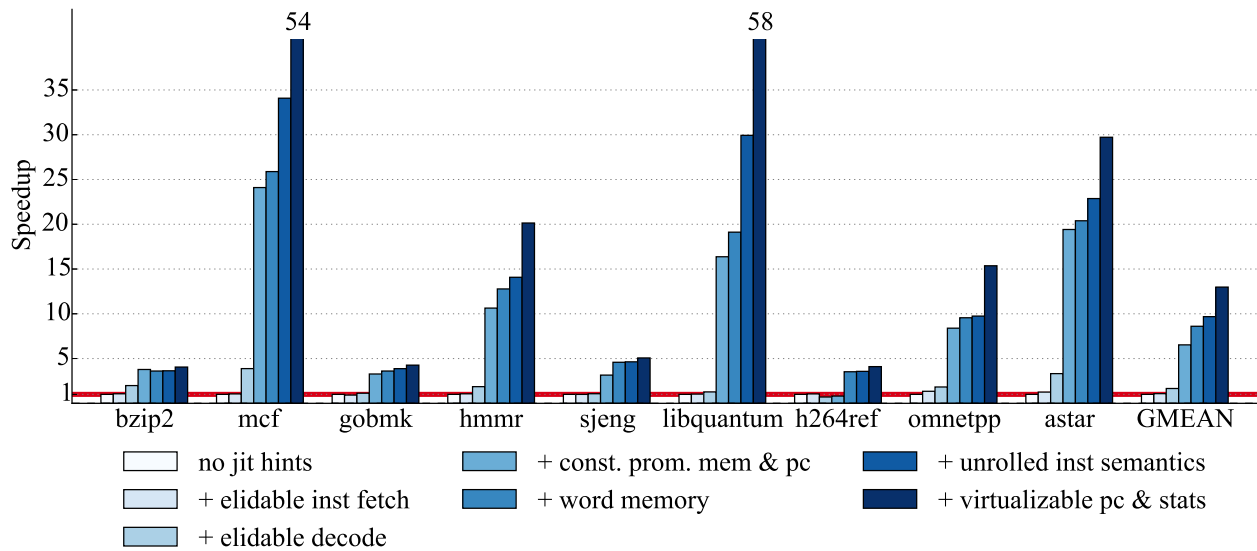
Figure 4.11: Impact of JIT Annotations – Including advanced annotations in the RPython interpreter allows our generated ISS to perform more aggressive JIT optimizations. However, the benefits of these optimizations varies from benchmark to benchmark. Above we show how incrementally combining several 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.

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, 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 26 in Figure 4.10, can potentially eliminate 10 JIT IR nodes on lines 1–4 in Figure 4.12, it shows negligible performance benefit in Figure 4.11. 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 4.12. The combination of elidable instruction fetch and elidable decode shows the first performance increase for many applications in Figure 4.11.

75

```
 1   # Byte accesses for instruction fetch
 2   i1 = getarrayitem_gc(p6, 33259)
 3   i2 = int_lshift(i1, 8)
 4   # ... 8 more JIT IR nodes ...
 5
 6   # Decode function call
 7   p1 = call(decode, i3)
 8   guard_no_exception()
 9   i4 = getfield_gc_pure(p1)
10   guard_value(i4, 4289648)
11
12   # Accessing instruction fields
13   i5 = int_rshift(i3, 28)
14   guard_value(i5, 14)
15   i6 = int_rshift(i3, 25)
16   i7 = int_and(i6, 1)
17   i8 = int_is_true(i7)
18   # ... 11 more JIT IR nodes ...
19
20   # Read from regfile
21   i10 = getarrayitem_gc(p2, i9)
22
23   # Register offset calculation
24   i11 = int_and(i10, 0xffffff)
25   i12 = int_rshift(i3, 16)
26   i13 = int_and(i12, 15)
27   i14 = int_eq(i13, 15)
28   guard_false(i14)
29
30   # Read from regfile
31   i15 = getarrayitem_gc(p2, i13)
```

```
32   # Addressing mode
33   i15 = int_rshift(i3, 23)
34   i16 = int_and(i15, 1)
35   i17 = int_is_true(i16)
36   # ... 13 more JIT IR nodes ...
37
38   # Access mem with byte reads
39   i19 = getarrayitem_gc(p6, i18)
40   i20 = int_lshift(i19, 8)
41   i22 = int_add(i21, 2)
42   # ... 13 more JIT IR nodes ...
43
44   # Write result to regfile
45   setarrayitem_gc(p2, i23, i24)
46
47   # Update PC
48   i25 = getarrayitem_gc(p2, 15)
49   i26 = int_add(i25, 4)
50   setarrayitem_gc(p2, 15, i26)
51   i27 = getarrayitem_gc(p2, 15)
52   i28 = int_lt(i27, 33256)
53   guard_false(i28)
54   guard_value(i27, 33260)
55
56   # Update cycle count
57   i30 = int_add(i29, 1)
58   setfield_gc(p0, i30)
```

Figure 4.12: 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.

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

Figure 4.13: 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.

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 37–39 in Figure 4.10. 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 4.12) can truly be removed from the optimized trace. Figure 4.11 shows how all three optimizations work together to increase performance by $5\times$ on average and up to $25\times$ 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 11–24 in Figure 4.10 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 4.12 into the single memory access on line 6 of Figure 4.13. The number and kind of memory accesses performed influence the benefits of this optimization. In Figure 4.11 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 4.11 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 (lines 2 and 5 of Figure 4.10) is able to eliminate lines 47–58 from Figure 4.12 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.

Maximum Trace Length   Although not shown in Figure 4.11, another optimization that must be specifically tuned for instruction set simulators is the maximum trace length threshold. The maximum trace length does not impact the quality of JIT compiled code as the previously discussed optimizations do, rather, it impacts *if* and *when* JIT compiliation occurs at all. This parameter determines how long of an IR sequence the JIT should trace before giving up its search for hot loops to optimize. Longer traces may result in performance degradation if they do not lead to the discovery of additional hot loops because the considerable overheads of tracing cannot be amortized unless they ultimately result in the generation of optimized, frequently executed assembly. Instructions executed by an instruction set simulator are simpler than the bytecode of a dynamic language,
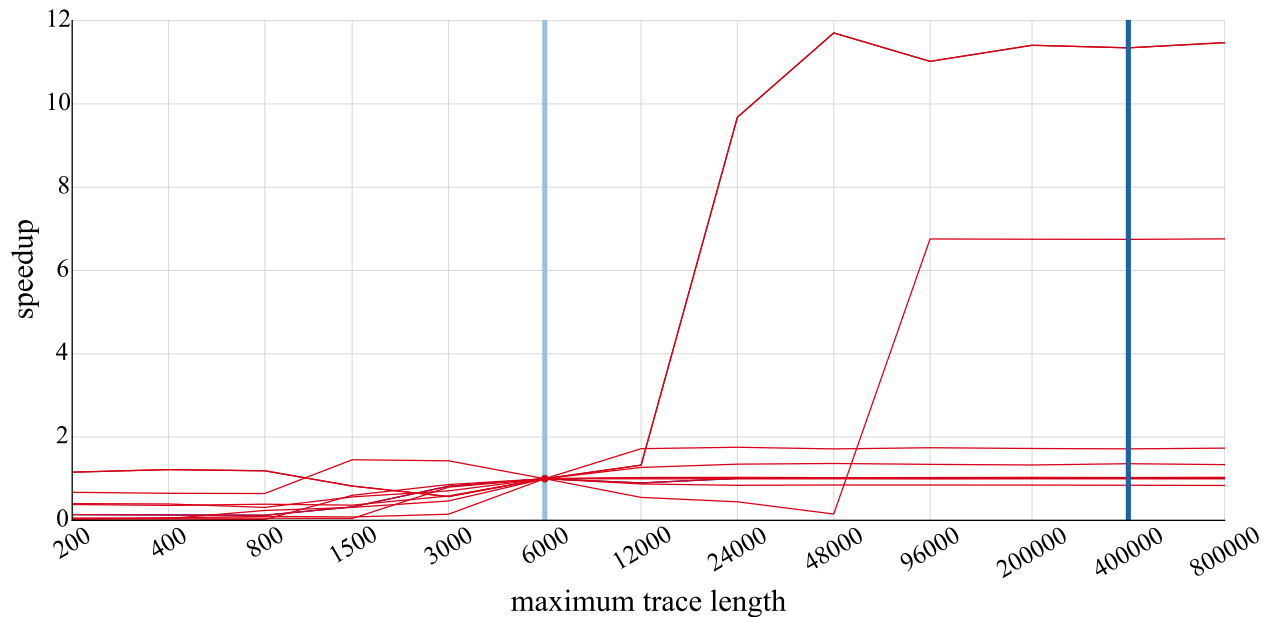
Figure 4.14: Impact of Maximum Trace Length – The plot above shows the performance impact of changing the JIT's maximum trace length in a Pydgin ISS. Each line represents a different SPEC CINT2006 benchmark; each benchmark is normalized to its own performance when using RPython translation toolchain's default maximum trace length of 6000 (light blue line). For some benchmarks, Pydgin ISSs may experience extremely poor performance when using this default trace length; occasionally this performance is even worse than a Pydgin ISS generated *without* the JIT at all. This was found to be true for the SPEC CINT2006 benchmarks *464.h264ref* and *401.bzip2*. Increasing the JIT's maximum trace length up to 400000 (dark blue line) resulted in considerable performance improvements in both of these benchmarks.

resulting in large traces that occassionally exceed the default threshold. Figure 4.14 demonstrates how the value of the maximum trace length threshold impacts Pydgin ISS performance for a number of SPEC CINT2006 benchmarks. Two benchmarks in particular, *464.h264ref* and *401.bzip2*, benefit considerably from a significantly larger threshold.

4.5 Performance Evaluation of Pydgin ISSs

We evaluate Pydgin by implementing three ISAs using the Pydgin embedded-ADL: a simplified version of MIPS32 (SMIPS), a subset of ARMv5, and RISC-V RV64G. These embedded-ADL descriptions are combined with RPython optimization annotations, including those described in Section 4.4, to generate high-performance, JIT-enabled DBT-ISSs. Traditional interpretive ISSs without JITs are also generated using the RPython translation toolchain in order to help quantify

Simulation Host	
CPU	Intel Xeon E5620
Frequency	2.40GHz
RAM	48GB @ 1066 MHz

Target Hosts			
ISA	Simplified MIPS32	ARMv5	RISC-V RV64G
Compiler	Newlib GCC 4.4.1	Newlib GCC 4.3.3	Newlib GCC 4.9.2
Executable	Linux ELF	Linux ELF	Linux ELF64
System Calls	Emulated	Emulated	Emulated or Proxied
Floating Point	Soft Float	Soft Float	Hard Float

Table 4.1: Simulation Configurations – All experiments were performed on an unloaded target machine described above. The ARMv5, Simplified MIPS (SMIPS), and RISC-V ISSs all used system call emulation, except for *spike* which used a proxy kernel. SPEC CINT2006 benchmarks were cross-compiled using SPEC recommended optimization flags (-O2). ARMv5 and SMIPS binaries were compiled to use software floating point.

the performance benefit of the meta-tracing JIT. We compare the performance of these Pydgin-generated ISSs against several reference ISSs.

To quantify the simulation performance of each ISS, we collected total simulator execution time and simulated MIPS metrics from the ISSs running SPEC CINT2006 applications. All applications were compiled using the recommended SPEC optimization flags (-O2) and all simulations were performed on unloaded host machines. SMIPS and ARMv5 binaries were compiled with software floating point enabled, while RISC-V binaries used hardware floating point instructions. Complete compiler and host-machine details can be found in Table 4.1. Three applications from SPEC CINT2006 (*400.perlbench*, *403.gcc*, and *483.xalancbmk*) would not build successfully due to limited system call support in our Newlib-based cross-compilers. When evaluating the high-performance DBT-ISSs, target applications were run to completion using datasets from the SPEC reference inputs. Simulations of most interpretive ISSs were terminated after 10 billion simulated instructions since the poor performance of these simulators would require many hours, in some cases days, to run these benchmarks to completion. Total application runtimes for the truncated simulations (labeled with *Time** in Tables 4.2, 4.3, and 4.4) were extrapolated using MIPS measurements and dynamic instruction counts. Experiments on a subset of applications verified the simulated MIPS computed from these truncated runs provided a good approximation of MIPS measurements collected from full executions. This matches prior observations that interpretive

ISSs demonstrate very little performance variation across program phases. Complete information on the SPEC CINT2006 application input datasets and dynamic instruction counts can be found in Tables 4.2, 4.3, and 4.4.

Reference simulators for SMIPS include a slightly modified version of the gem5 MIPS atomic simulator (*gem5-smips*) and a hand-written C++ ISS used internally for teaching and research purposes (*cpp-smips*). Both of these implementations are purely interpretive and do not take advantage of any JIT-optimization strategies. Reference simulators for ARMv5 include the gem5 ARM atomic simulator (*gem5-arm*), interpretive and JIT-enabled versions of SimIt-ARM (*simit-arm-nojit* and *simit-arm-jit*), as well as QEMU. Atomic models from the gem5 simulator [BBB⁺11] were chosen for comparison due their wide usage amongst computer architects. SimIt-ARM [DQ06, QDZ06] was selected because it is currently the highest performance ADL-generated DBT-ISS publicly available. QEMU has long been held as the gold-standard for DBT simulators due to its extremely high performance [Bel05]. Note that QEMU achieves its excellent performance at the cost of observability. Unlike QEMU, all other simulators in this study faithfully track architectural state at an instruction level rather than block level. The only reference simulator used to compare RISC-V was *spike*, a hand-written C++ ISS and golden model for the RISC-V ISA specification.

4.5.1 SMIPS

Table 4.2 shows the complete performance evaluation results for each SMIPS ISS while Figure 4.15 shows a plot of simulator performance in MIPS. Pydgin's generated interpretive and DBT-ISSs are able to outperform *gem5-smips* and *cpp-smips* by a considerable margin: around a factor of 8–9× for *pydgin-smips-nojit* and a factor of 25–200× for *pydgin-smips-jit*. These speedups translate into considerable improvements in simulation times for large applications in SPEC CINT2006. For example, whereas *471.omnetpp* would have taken eight days to simulate on *gem5-smips*, this runtime is drastically reduced down to 21.3 hours on *pydgin-smips-nojit* and an even more impressive 1.3 hours on *pydgin-smips-jit*. These improvements significantly increase the applications researchers can experiment with when performing design-space exploration.

The interpretive ISSs tend to demonstrate relatively consistent performance across all benchmarks: 3–4 MIPS for *gem5-smips* and *cpp-smips*, 28–36 MIPS for *pydgin-smips-nojit*. Unlike DBT-ISSs, which optimize away many overheads for frequently encountered instruction paths, interpretive ISSs must perform both instruction fetch and decode for every instruction simulated.

81
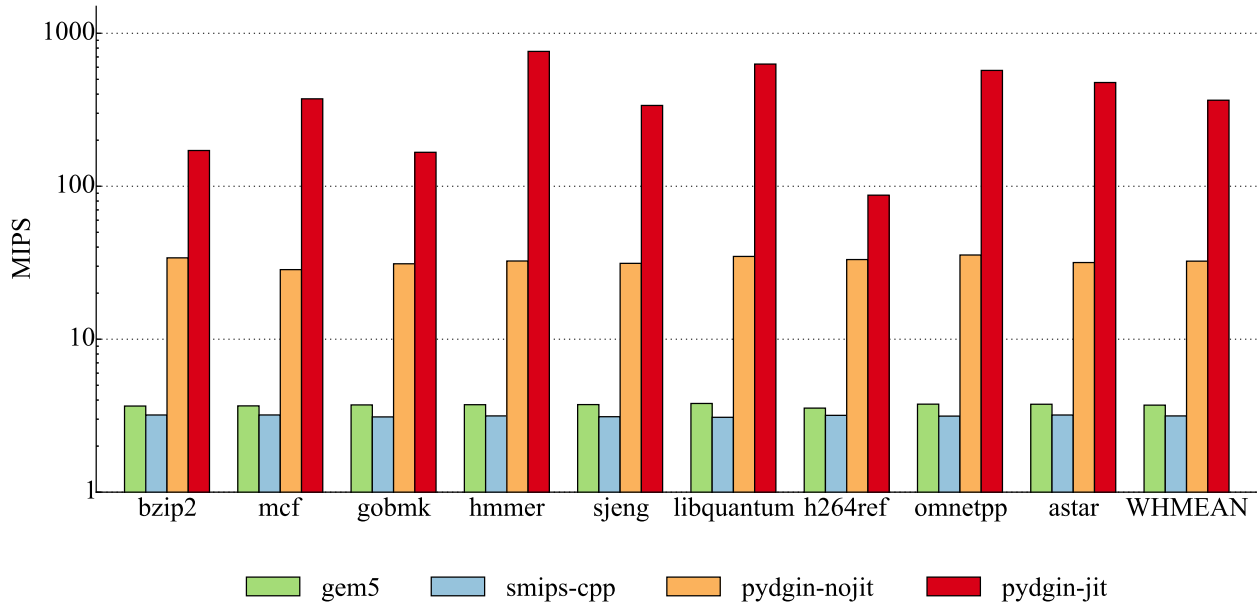
Figure 4.15: SMIPS Instruction Set Simulator Performance

These overheads limit the amount of simulation time variability, which is primarily caused by complexity differences between instruction implementations.

Also interesting to note are the different implementation approaches used by each of these interpretive simulators. The *cpp-smips* simulator is completely hand-coded with no generated components, whereas the *gem5-smips* decoder and instruction classes are automatically generated from what the gem5 documentation describes as an "ISA description language" (effectively an ad-hoc and relatively verbose ADL). As mentioned previously, *pydgin-smips-nojit* is generated from a high-level embedded-ADL. Both the generated *gem5-smips* and *pydgin-smips-nojit* simulators are able to outperform the hand-coded *cpp-smips*, demonstrating that generated simulator approaches can provide both productivity and performance advantages over simple manual implementations.

In addition to providing significant performance advantages over *gem5-smips*, both Pydgin simulators provide considerable productivity advantages as well. Because the gem5 instruction descriptions have no interpreter, they must be first generated into C++ before testing. This leaves the user to deduce whether the source of an implementation bug resides in the instruction definition, the code generator, or the gem5 simulator framework. In comparison, Pydgin's embedded-ADL is fully compliant Python that requires no custom parsing and can be executed directly in a standard Python interpreter. This allows Pydgin ISA implementations to be tested and verified using Python

Benchmark	Dataset	Inst (B)	gem5		cpp			pydgin nojit			pydgin jit		
			Time*	MIPS	Time*	MIPS	vs. g5	Time*	MIPS	vs. g5	Time	MIPS	vs. g5
401.bzip2	chicken.jpg	198	15.1h	3.7	17.2h	3.2	0.87	1.6h	34	9.3	19.3m	171	47
429.mcf	inp.in	337	1.1d	3.7	1.2d	3.2	0.87	3.3h	28	7.8	15.0m	373	102
445.gobmk	13x13.tst	290	21.7h	3.7	1.1d	3.1	0.83	2.6h	31	8.4	29.0m	167	45
456.hmmer	nph3.hmm	1212	3.8d	3.7	4.5d	3.2	0.84	10.4h	32	8.7	26.5m	761	204
458.sjeng	ref.txt	2757	8.5d	3.7	10.2d	3.1	0.83	1.0d	31	8.4	2.3h	337	90
462.libquantum	1397 8	2917	8.9d	3.8	10.9d	3.1	0.81	23.3h	35	9.1	1.3h	629	165
464.h264ref	foreman_ref	679	2.2d	3.5	2.5d	3.2	0.90	5.7h	33	9.4	2.2h	87	25
471.omnetpp	omnetpp.ini	2708	8.3d	3.8	10.0d	3.1	0.84	21.2h	36	9.4	1.3h	572	152
473.astar	BigLakes2048.cfg	472	1.5d	3.8	1.7d	3.2	0.85	4.1h	32	8.4	16.5m	476	127

Table 4.2: Detailed SMIPS Instruction Set Simulator Performance Results – Benchmark datasets taken from the SPEC CINT2006 reference inputs. Time is provided in either minutes (m), hours (h), or days (d) where appropriate. Time* indicates runtime estimates that were extrapolated from simulations terminated after 10 billion instructions. DBT-ISSs (*pydgin-smips-jit*) were simulated to completion. vs. g5 = simulator performance normalized to gem5.

debugging tools prior to RPython translation into a fast C implementation, leading to a much more user-friendly debugging experience.

Enabling JIT optimizations in the RPython translation toolchain results in a considerable improvement in Pydgin-generated ISS performance: from 28–36 MIPS for *pydgin-smips-nojit* up to 87–761 MIPS for *pydgin-smips-jit*. Compared to the interpretive ISSs, *pydgin-smips-jit* demonstrates a much greater range of performance variability that depends on the characteristics of the application being simulated. The RPython generated meta-tracing JIT is designed to optimize hot loops and performs best on applications that execute large numbers of frequently visited loops with little branching behavior. As a result, applications with large amounts of irregular control flow cannot be optimized as well as more regular applications. For example, although *445.gobmk* shows decent speedups on *pydgin-smips-jit* when compared to the interpretive ISSs, its performance in MIPS lags that of some other applications. Some of this performance lag is due to the use of the default maximimum trace length threshold; as discussed in Section 4.4 increasing this threshold should greatly improve the performance of benchmarks like *464.h264ref*. However, improving DBT-ISS performance on challenging irregular applications is still an open area of research in the tracing-JIT community.
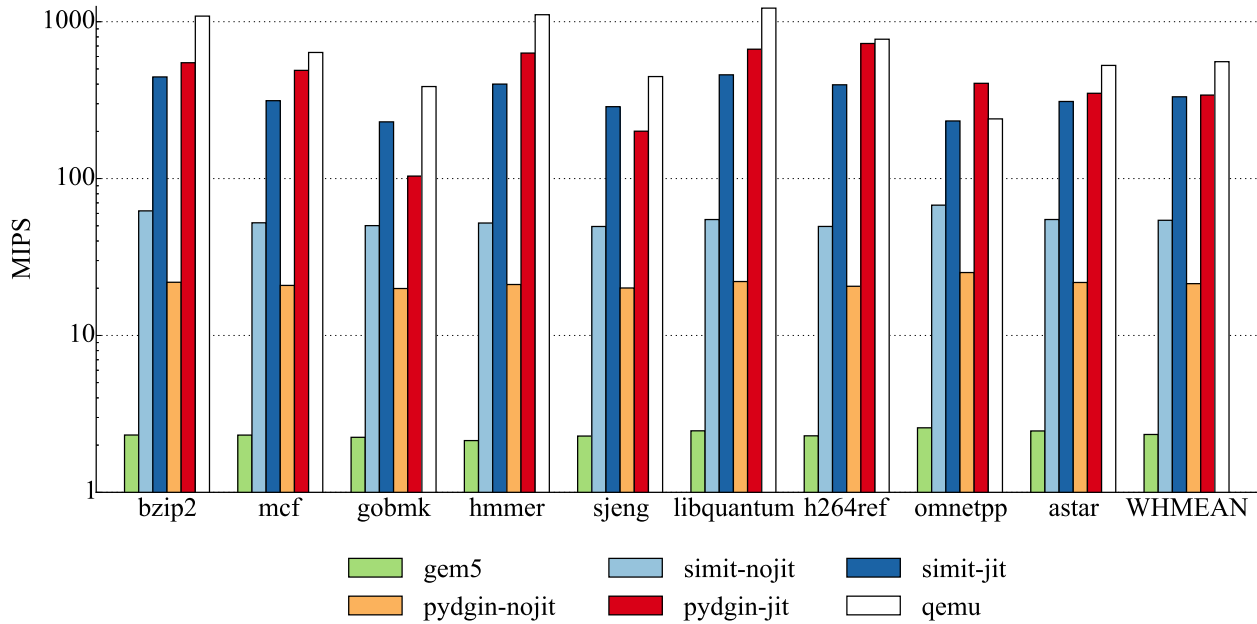
Figure 4.16: ARMv5 Instruction Set Simulator Performance

4.5.2 ARMv5

The ARMv5 ISA demonstrates significantly more complex instruction behavior than the relatively simple SMIPS ISA. Although still a RISC ISA, ARMv5 instructions include a number of interesting features that greatly complicate instruction processing such as pervasive use of conditional instruction flags and fairly complex register addressing modes. This additional complexity makes ARMv5 instruction decode and execution much more difficult to emulate efficiently when compared to SMIPS. This is demonstrated in the relative performance of the two gem5 ISA models shown in Tables 4.2 and 4.3: *gem5-arm* performance never exceeds 2.6 MIPS whereas *gem5-smips* averages 3.7 MIPS. Note that this trend is also visible when comparing *pydgin-arm-nojit* (20–25 MIPS) and *pydgin-smips-nojit* (28–36 MIPS). Complete performance results for all ARMv5 ISSs can be found in Table 4.3 and Figure 4.16.

To help mitigate some of the additional decode complexity of the ARMv5 ISA, ISS implementers can create more optimized instruction definitions that deviate from the pseudo-code form described in the ARMv5 ISA manual (as previously discussed in Section 4.3). These optimizations and others enable the SimIt-ARM ISS to achieve simulation speeds of 49–68 MIPS for *simit-arm-nojit* and 230–459 MIPS for *simit-arm-jit*. In comparison, Pydgin's more straightforward ADL

84

Benchmark	Inst (B)	gem5 Time*	gem5 MIPS	simit nojit Time*	simit nojit MIPS	simit nojit vs. g5	simit jit Time	simit jit MIPS	simit jit vs. g5	pydgin nojit Time*	pydgin nojit MIPS	pydgin nojit vs. g5	pydgin nojit vs. s0	pydgin jit Time	pydgin jit MIPS	pydgin jit vs. g5	pydgin jit vs. sJ	qemu Time	qemu MIPS
401.bzip2	195	23.4h	2.3	52.2m	62	27	7.3m	445	192	2.5h	22	9.4	0.35	5.9m	548	236	1.23	3.0m	1085
429.mcf	374	1.9d	2.3	2.0h	52	23	19.9m	314	135	5.0h	21	9.0	0.40	12.7m	489	211	1.56	9.8m	637
445.gobmk	324	1.7d	2.2	1.8h	50	22	23.5m	230	103	4.5h	20	8.9	0.40	52.0m	104	46	0.45	14.0m	386
456.hmmer	1113	6.0d	2.1	5.9h	52	24	46.3m	400	187	14.6h	21	9.9	0.41	29.4m	631	295	1.58	16.7m	1108
458.sjeng	2974	15.1d	2.3	16.7h	49	22	2.9h	287	126	1.7d	20	8.8	0.41	4.1h	200	88	0.70	1.8h	447
462.libquantum	3070	14.4d	2.5	15.6h	55	22	1.9h	459	186	1.6d	22	8.9	0.40	1.3h	668	271	1.46	41.9m	1220
464.h264ref	753	3.8d	2.3	4.2h	50	22	31.7m	396	173	10.2h	21	9.0	0.42	17.3m	726	317	1.83	16.2m	773
471.omnetpp	1282	5.8d	2.6	5.3h	68	26	1.5h	233	90	14.1h	25	9.8	0.37	52.8m	405	157	1.74	1.5h	240
473.astar	434	2.0d	2.5	2.2h	55	22	23.3m	310	126	5.5h	22	8.8	0.40	20.7m	350	142	1.13	13.8m	526

Table 4.3: Detailed ARMv5 Instruction Set Simulator Performance Results – Benchmark datasets taken from the SPEC CINT2006 reference inputs (shown in Table 4.2). Time is provided in either minutes (m), hours (h), or days (d) where appropriate. Time* indicates runtime estimates that were extrapolated from simulations terminated after 10 billion instructions. DBT-ISSs (*simit-arm-jit*, *pydgin-arm-jit*, and QEMU) were simulated to completion. vs. g5 = simulator performance normalized to gem5. vs. s0 = simulator performance normalized to *simit-arm-nojit*. vs. sJ = simulator performance normalized to *simit-arm-jit*.

descriptions of the ARMv5 ISA result in an ISS performance of 20–25 MIPS for *pydgin-arm-nojit* and 104–726 MIPS for *pydgin-arm-jit*.

Comparing the interpretive versions of the SimIt-ARM and Pydgin generated ISSs reveals that *simit-arm-nojit* is able to outperform *pydgin-arm-nojit* by a factor of 2× on all applications. The fetch and decode overheads of interpretive simulators make it likely much of this performance improvement is due to SimIt-ARM's decode optimizations. However, decode optimizations should have less impact on DBT-ISSs which are often able to eliminate decode entirely.

The DBT-ISS versions of SimIt-ARM and Pydgin exhibit comparatively more complex performance characteristics: both *pydgin-arm-jit* and *simit-arm-jit* are able to provide good speedups across all applications, however, *pydgin-arm-jit* has a greater range in performance variability (104–726 MIPS for *pydgin-arm-jit* compared to 230–459 MIPS for *simit-arm-jit*). Overall *pydgin-arm-jit* is able to outperform *simit-arm-jit* on seven out of nine applications, speedups for these benchmarks ranged from 1.13–1.83×. The two underperforming benchmarks, *445.gobmk* and *458.sjeng*, observed slowdowns of 0.45× and 0.70× compared to *simit-arm-jit*. Despite the exceptional top-end performance of *pydgin-arm-jit* (726 MIPS) and its ability to outperform *simit-arm-jit* on all but two benchmarks tested, it only slightly outperformed *simit-arm-jit* when comparing

the weighted-harmonic mean results for the two simulators: 340 MIPS versus 332 MIPS. This is largely due to the poor performance on *458.sjeng*, which is a particularly large benchmark both in terms of instructions and runtime.

The variability differences displayed by these two DBT-ISSs is a result of the distinct JIT architectures employed by Pydgin and SimIt-ARM. Unlike *pydgin-arm-jit*'s meta-tracing JIT which tries to detect hot loops and highly optimize frequently taken paths through them, *simit-arm-jit* uses a page-based approach to JIT optimization that partitions an application binary into equal sized bins, or *pages*, of sequential program instructions. Once visits to a particular page exceed a preset threshold, all instructions within that page are compiled together into a single optimized code block. A page-based JIT provides two important advantages over a tracing JIT: first, pages are constrained to a fixed number of instructions (on the order of 1000) which prevents unbounded trace growth for irregular code; second, pages enable JIT-optimization of code that does not contain loops. While this approach to JIT design prevents SimIt-ARM from reaching the same levels of optimization as a trace-based JIT on code with regular control flow, it allows for more consistent performance across a range of application behaviors.

QEMU also demonstrates a wide variability in simulation performance depending on the application (240–1220 MIPS), however it achieves a much higher maximum performance and manages to outperform *simit-arm-jit* and *pydgin-arm-jit* on nearly every application except for *471.omnetpp*. Although QEMU has exceptional performance, it has a number of drawbacks that impact its usability. Retargeting QEMU simulators for new instructions requires manually writing blocks of low-level code in the tiny code generator (TCG) intermediate representation, rather than automatically generating a simulator from a high-level ADL. Additionally, QEMU sacrifices observability by only faithfully tracking architectural state at the block level rather than at the instruction level. These two limitations impact the productivity of researchers interested in rapidly experimenting with new ISA extensions.

4.5.3 RISC-V

RISC-V is another RISC-style ISA that lies somewhere between SMIPS and ARMv5 in terms of complexity. The decoding and addressing modes of RISC-V are considerably simpler than ARMv5, however, RISC-V has several extensions that add instructions for advanced features like atomic memory accesses and floating-point operations. Our RISC-V ISS implements the RV64G
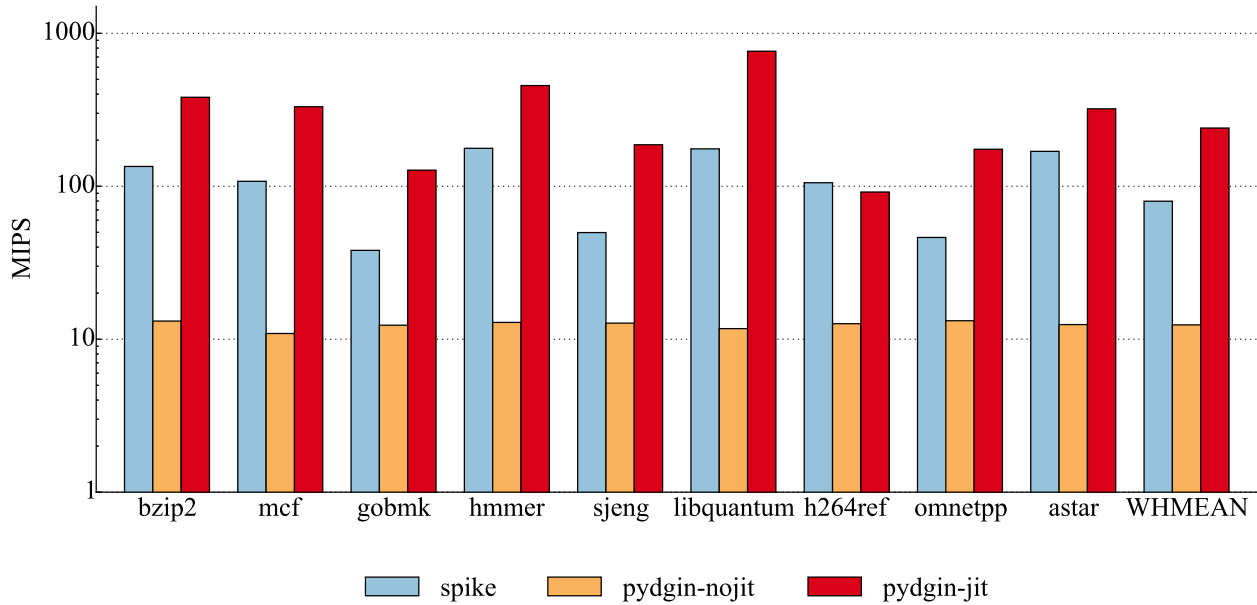
Figure 4.17: RISC-V Instruction Set Simulator Performance

variant of the RISC-V ISA; this includes 64-bit instructions and the full collection of "standard" extensions: integer multiplication and division (M), atomics (A), single-precision floating-point (F), and double-precision floating-point (D). The Pydgin SMIPS and ARMv5 ISSs discussed previously implement only 32-bit instructions.

One significant challenge of building the RISC-V ISS was implementing the full collection of single- and double-precision floating-point instructions defined in the RISC-V F and D extensions. The SMIPS and ARMv5 ISSs currently only include basic conversion instructions between floating-point and integer register values; these conversions are relatively simple to perform in RPython. IEEE-compliant arithmetic floating-point instructions are much more complex and require extensive logic to handle rounding modes and overflow behavior. Rather than re-writing this complex logic in Pydgin, existing C implementations provided by the open-source *softfloat* library were used via a foreign-function interface (FFI). Ensuring these FFI calls could be translated by the RPython toolchain was non-trivial, however, once this work was completed softfloat was fully wrapped in a Pydgin library and available for reuse in future ISSs.

The simulation performance of *pydgin-riscv-nojit*, *pydgin-riscv-jit*, and *spike* can be seen in Figure 4.17 and Table 4.4. The *spike* simulator demonstrates impressive performance that ranges from 38–177 MIPS. Although *spike* does not advertise itself as a DBT-ISS, its performance variation hints at the use of JIT-optimization strategies; comparatively, purely interpretive ISSs show

87

Benchmark	Dataset	spike			pydgin nojit			pydgin jit			
		Inst (B)	Time	MIPS	Time*	MIPS	vs. sp	Inst (B)	Time	MIPS	vs. sp
401.bzip2	chicken.jpg	202	25.0m	135	4.3h	13	0.10	203	8.9m	382	2.8
429.mcf	inp.in	292	45.1m	108	7.4h	11	0.10	290	14.6m	331	3.1
445.gobmk	13x13.tst	271	2.0h	38	6.1h	12	0.32	271	35.5m	127	3.3
456.hmmer	nph3.hmm	966	1.5h	177	20.9h	13	0.07	967	35.4m	455	2.6
458.sjeng	ref.txt	2852	15.9h	50	2.7d	13	0.26	2953	4.4h	187	3.8
462.libquantum	1397 8	2141	3.4h	176	2.1d	12	0.07	2142	46.8m	763	4.3
464.h264ref	foreman_ref	606	1.6h	105	13.4h	13	0.12	609	1.8h	92	0.87
471.omnetpp	omnetpp.ini	561	3.4h	46	11.7h	13	0.29	558	53.3m	174	3.8
473.astar	rivers.cfg	801	1.3h	169	17.9h	12	0.07	803	41.7m	321	1.9

Table 4.4: Detailed RISC-V Instruction Set Simulator Performance Results – Benchmark datasets taken from the SPEC CINT2006 reference inputs. Time is provided in either minutes (m), hours (h), or days (d) where appropriate. Time* indicates runtime estimates that were extrapolated from simulations terminated after 10 billion instructions. Both *spike* and *pydgin-riscv-jit* were executed to completion, but instruction counts differ since *spike* uses a proxy kernel rather than syscall emulation. vs. sp = simulator performance normalized to *spike*.

fairly consistent MIPS from application to application. Some optimizations implemented in *spike* which are potentially responsible for this variability include the caching of decoded instruction execution functions and the aggressive unrolling of the interpreter loop into a switch statement with highly predictable branching behavior. The interpretive *pydgin-riscv-nojit* only achieves 11–13 MIPS, which is significantly slower than *spike*. It is also surprisingly slower than both *pydgin-nojit-smips* and *pydgin-nojit-arm* by a fair margin; this is likely related to the use of hardware floating-point in the RISC-V binaries rather than software floating-point and the fact that these floating-point instructions are implemented using FFI calls.

The JIT-enabled *pydgin-riscv-jit* handily outperforms *spike* with a weighted-harmonic mean performance of 270 MIPS versus 80 MIPS. The one exception where *spike* outperformed *pydgin-riscv-jit* was *464.h264ref*; this is because the maximum trace length threshold was not changed from the default value. As shown in Section 4.4, increasing this threshold value should significantly improve the performance of *464.h264ref* on *pydgin-riscv-jit*. Excluding *464.h264ref*, the performance of *pydgin-riscv-jit* ranged from 127–763 MIPS. This impressive performance was achieved in fewer than two weeks of work, clearly demonstrating the power of the Pydgin framework.
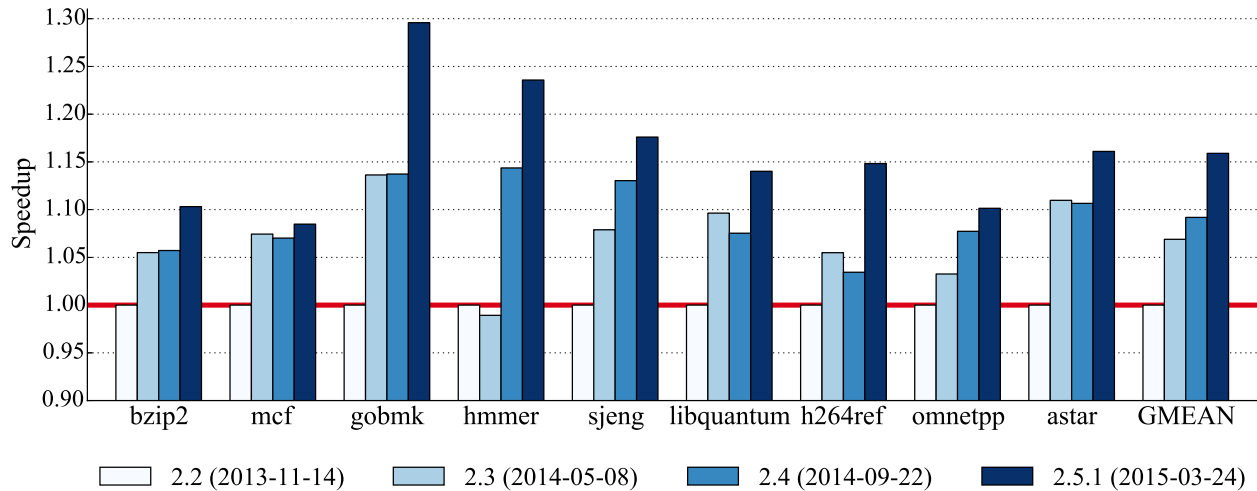
Figure 4.18: RPython Performance Improvements Over Time – Pydgin instruction set simulators can benefit from enhancements made to the RPython translation toolchain's meta-tracing JIT compiler by simply downloading updated releases and recompiling. The impact of these RPython enhancements on the performance of the `pydgin-arm-jit` ISS are shown above. Four versions of `pydgin-arm-jit` were generated from different releases of the RPython translation toolchain from 2.2–2.5.1; these releases, occuring over a 16 month time span, incrementally led to a 16% performance boost on the SPEC benchmark suite.

4.5.4 Impact of RPython Improvements

A considerable benefit of building Pydgin on top of the RPython translation toolchain is that it benefits from improvements made to the toolchain's JIT compiler generator. PyPy and the RPython translation toolchain used to construct PyPy are both open-source projects under heavy active development. Several of the developers are experts in the field of tracing-JIT compilers, these developers often use RPython as a platform to experiment with optimizations for Python and other languages. Successful optimizations are frequently integrated into the PyPy and RPython source code and are made available to the public via regular version releases.

Figure 4.18 shows how these advances impact the performance of Pydgin instruction set simulators. The `pydgin-arm-jit` ISS was recompiled with several snapshots of the RPython translation toolchain that come distributed with PyPy releases. Each successive release we tested since 2.2 resulted in an overall performance improvement across the benchmarks, although some individual benchmarks saw minor performance regressions between versions. A Pydgin user upgrading their RPython toolchain from version 2.2 to 2.5.1, released approximately 16 months apart, would achieve an impressive 16% performance improvement on our benchmarks by simply recompiling.

4.6 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. Prior work also exists on instruction-set simulators that utilize static binary translation, such as SyntSim [BG04]; a primary limitation of such static approaches is their inability to easily handle self-modifying code.

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], and hybrid-compiled [RMD09, RD03]. More recently, dynamic-compiled [QDZ06, BFKR09, PKHK11], multicore/distributed dynamic-compiled [DQ06], and parallel DBT [WGFT13] have become the more popular techniques.

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.

4.7 Conclusion

In an era of rapid development of increasingly specialized system-on-chip platforms, instruction set simulators can sacrifice neither designer productivity nor simulation performance. However, constructing ISS toolchains that are both highly productive and high performance remains a significant research challenge. Pydgin as been introduced as a novel approach to address these multiple challenges by enabling the automatic generation of high-performance DBT-ISSs from a Python-based embedded-ADL. Pydgin's automatic DBT-generation capabilities are made possible by creatively adapting the RPython translation toolchain, an existing meta-tracing JIT compilation framework designed for general-purpose dynamic programming languages.

The Pydgin framework and the Pydgin SMIPS, ARMv5, and RISC-V ISSs have been publicly released under an open-source software license. My hope is that the productivity and performance benefits of Pydgin will make it a useful framework for the broader computer architecture research community and beyond.

CHAPTER 5
EXTENDING THE SCOPE OF VERTICALLY INTEGRATED DESIGN IN PYMTL

The previous two chapters introduced PyMTL and Pydgin, two frameworks I have developed to simplify the process of hardware modeling. PyMTL and Pydgin were created in order to facilitate the implementation of three particular model classes widely used in computer architecture research: functional-, cycle-, and register-transfer level models. While Pydgin was designed to ease the implementation of a specific, performance-critical class of FL models (i.e., instruction set simulators), PyMTL was intended to be a more general-purpose framework that enables the creation of many different types of hardware models.

With that in mind, this chapter explores additional mechanisms that further extend the capabilities of PyMTL for hardware design. The first mechanism, high-level synthesis (HLS), is a technique for reducing the complexity of RTL design through the direct, software automated conversion of FL models into RTL models. The second mechanism, gate-level (GL) modeling and floorplanning, aims to extend PyMTL's modeling capabilities down the stack into the realm of physical design. Both mechanisms aim to augment and enhance the modeling towards layout philosophy of PyMTL by providing a unified environment that fosters many different modeling styles and hardware implementation techniques.

5.1 Transforming FL to RTL: High-Level Synthesis in PyMTL

High-level synthesis (HLS) is an RTL design process which takes a high-level algorithm, usually in C++ or SystemC, and attempts to infer a hardware-implementation based on the algorithm structure and user-provided annotations. Due to the rapid improvement in HLS algorithms and the growing demand for specialization, HLS is becoming increasingly popular as a means to accelerate RTL development and as a tool for rapid design-space exploration. A comparison of HLS with the *modeling towards layout* (MTL) methodology promoted by PyMTL reveals that HLS and MTL are orthogonal approaches to hardware design: MTL encourages the *manual*, incremental refinement of a design from FL, to CL, to RTL, whereas HLS enables an *automatic* transformation from FL to RTL. These methodologies each have their strengths and weaknesses, and the more productive approach largely depends on the design task at hand.

In this section, I describe some basic experiments with a prototype tool for performing high-level synthesis on PyMTL FL models. I see this as a promising future direction for a number of reasons. Supporting both HLS and MTL design methodologies within PyMTL would allow designers to choose the most suitable design approach on a component by component basis, without requiring the need to change development environments. Unifying HLS and MTL under PyMTL would also address many of the integration problems associated with current HLS toolflows: HLS tools cannot easily utilize user-provided infrastructure for automatic verification within existing test benches or for co-simulation with existing FL, CL, and RTL models. Finally, the more abstract algorithm implementations provided by productivity-level languages like Python may provide additional information to HLS tools, enabling powerful scheduling and logic optimizations not inferable from equivalent C++ code.

5.1.1 HLSynthTool Design

An experimental PyMTL tool called `HLSynthTool` was created to enable the automated generation of RTL from PyMTL functional-level models. Rather than implementing a full Python-to-HDL synthesis flow from scratch, `HLSynthTool` converts PyMTL FL code into Vivado HLS compatible C++. This C++ code can then be passed into the Xilinx Vivado HLS tool in order to generate Verilog RTL. Currently this flow is not entirely automated: `HLSynthTool` generates only the input C++. Users are responsible for manually creating the TCL script driver for Vivado HLS and the `VerilogModel` wrapper for testing the Vivado HLS generated Verilog within PyMTL. Automatic generation of these components is future work; however, creation of a fully automated flow encapsulated in PyMTL may not be desirable due to the non-negligible latency of the Vivado HLS synthesis process.

The `HLSynthTool` tool prototype currently only supports a very limited subset of PyMTL FL code for translation into C++. Specifically, it requires FL models to use the `FLQueueAdapter` interface proxy to communicate with external models. The `FLQueueAdapter` interface proxy is designed to have blocking semantics. This allows designers writing FL models to implement internal logic in a straightforward manner without needing to worry about latency-insensitive communication protocols or concurrency. These interface proxies are translated into the Vivado HLS `hls::stream` class, which also has blocking semantics. All complexity of converting these blocking, method-based interfaces into synthesizable hardware is contained within the Vivado HLS tool.

5.1.2 Synthesis of PyMTL FL Models

An example PyMTL FL model that can be converted to Verilog RTL via high-level synthesis is shown in Figure 5.1. Note that this model is a normal PyMTL FL model and does not require any modification to force compatibility with the `HLSynthTool`. This is made possible by the explicit model/tool split provided by the PyMTL framework. Also note that this FL implementation makes use of the blocking, queue-based port adapters. These adapters proxy method-calls into latency-insensitive communication over the model's port interface. In addition to hiding the complexity of port-based communication protocols, these adapters significantly simplify the configuration logic for setting up the greatest-common divisor (GCD) accelerator. Concurrent execution semantics are hidden by the blocking behavior of the proxy method calls.

Once the correctness of this GcdXcelFL model has been verified using the usual `py.test` testing flow, the elaborated model can be passed into the `HLSynthTool` for conversion into Vivado HLS compatible C++. This generated-C++ is passed into Vivado HLS along with a TCL script for specifying synthesis parameters; the output of Vivado HLS is Verilog RTL. `VerilogModel` wrappers are used to wrap Verilog source in PyMTL port-based interfaces. This Vivado HLS generated, PyMTL-wrapped Verilog can now be imported and validated using the same `py.test` testing harness mentioned previously. With the addition of the `HLSynthTool` flow, PyMTL now has three ways to generate and interact with Verilog: (1) writing PyMTL RTL that is translated into Verilog HDL; (2) writing Verilog directly and importing it into PyMTL using a `VerilogModel` wrapper; and (3) generating RTL from PyMTL FL using the `HLSynthTool` and importing it with `VerilogModel`. Future work aims to further automate this process so that it becomes possible to verify the correctness of HLS-generated RTL with a simple command-line flag.

5.1.3 Algorithmic Experimentation with HLS

In order to demonstrate the opportunities for high-level synthesis in the context of PyMTL, the functional-level accelerator model shown in Figure 5.1 was used to explore several algorithms for computing the greatest-common divisor (GCD) of two integers. For this experiment, the accelerator configuration logic of the model in lines 8–44 is unchanged; only the implementation of the GCD function itself in lines 1–7 is modified. The benefit of this approach is that it allows the user to easily tweak the core algorithm in Python and experiment with various optimizations, verify the

```
1  # GCD algorithm under test.
2  def gcd( a, b ):
3    while a != b:
4      if ( a > b ): a = a - b
5      else:         b = b - a
6    return a
7
8  # Functional-level greatest-common divisor accelerator model.
9  class GcdXcelFL( Model ):
10
11   def __init__( s ):
12
13     s.xcelreq  = InValRdyBundle ( XcelReqMsg()  )
14     s.xcelresp = OutValRdyBundle( XcelRespMsg() )
15
16     s.req_queue  = InQueuePortProxy ( s.xcelreq  )
17     s.resp_queue = OutQueuePortProxy( s.xcelresp )
18
19     @s.tick_fl
20     def PyGcdXcelHLS():
21
22       # receive message containing data A, send ack
23       req  = s.req_queue.popleft()
24       a    = req.data
25       resp = XcelRespMsg().mk_msg( req.opaque, req.type_, 0, req.id )
26       s.resp_queue.append( resp )
27
28       # receive message containing data B, send ack
29       req  = s.req_queue.popleft()
30       b    = req.data
31       resp = XcelRespMsg().mk_msg( req.opaque, req.type_, 0, req.id )
32       s.resp_queue.append( resp )
33
34       # receive message containing go signal, send ack
35       req  = s.req_queue.popleft()
36       resp = XcelRespMsg().mk_msg( req.opaque, req.type_, 0, req.id )
37       s.resp_queue.append( resp )
38
39       result   = gcd( a, b )
40
41       # receive message requesting result, send ack
42       req  = s.req_queue.popleft()
43       resp = XcelRespMsg().mk_msg( req.opaque, req.type_, result, req.id )
44       s.resp_queue.append( resp )
```

Figure 5.1: PyMTL GCD Accelerator FL Model – An example PyMTL FL model that can automatically be translated to Verilog HDL using the experimental `HLSynthTool` and the Xilinx Vivado HLS tool. Proxy interfaces provide a programmer-friendly, method-based interface with blocking semantics. These interfaces are translated into blocking `hls::stream` classes provided by the Vivado HLS library.

```
1  def gcd( a, b ):          1  def gcd( a, b ):          1  def gcd( a, b ):
2    while b != 0:            2    while b != 0:            2    while a != b:
3      if a < b:              3                             3      if a > b:
4        # a, b = b, a        4      # a, b = b, a % b       4        a = a - b
5        t = b; b = a; a = t  5      t = b; b = a % b; a = t 5      else:
6      a = a - b              6                             6        b = b - a
7    return a                 7    return a                 7    return a
```

<div align="center">

(a)   (b)   (c)

</div>

Figure 5.2: Python GCD Implementations – Three distinct implementations of the GCD algorithm in Python. The PyMTL GCD Accelerator FL model shown in Figure 5.1 was augmented with each of these implementations and passed into the prototype `HLSynthTool`. Although Python enables concise swap operations without temporaries using tuple syntax (see commented lines above), `HLSynthTool` does not currently support this and users must rewrite such logic using temporary variables. Performance characteristics of hardware synthesized from these implementations are shown in Table 5.1.

correctness of these algorithms using existing `py.test` harnesses, and then obtain performance estimates for hardware implementations of these algorithms using `HLSynthTool`. This process enables rapid iteration of algorithmic-level optimizations without requiring an investment to manually implement RTL for each alternative. This is particularly advantageous when one considers that the greatest leverage for achieving large performance improvements generally comes from more efficient algorithms rather than from optimizations further down the stack.

Three different Python implementations of the GCD algorithm, shown in Figure 5.2, were used to perform the `GcdXcelFL` accelerator GCD computation on line 39 of Figure 5.1. These three implementations were passed into the `HLSynthTool` and targetted for synthesis in a Xilinx Zynq-7020 with a target clock frequency of 5 nanoseconds. The performance of each of these implementations as reported by simulations of `HLSynthTool`-generated RTL and synthesis reports emitted by Vivado HLS are shown in Table 5.1. Despite minor differences in the algorithms, there is considerable variance in the performance of the three implementations. Algorithm (b) has by far the worst timing, area, and performance characteristics by a large margin. This is because the modulo operator used by this algorithm on line 5 in Figure 5.2b gets synthesized into a complex divider unit which the other two implementations do not need. Algorithms (a) and (c), shown in Figures 5.2a and 5.2b, have identical timing and very similar characteristics, although (a) uses slightly more flip-flops (FFs) and fewer look-up tables (LUTS) than (c). However, (c) has considerably better execution performance than (a) demonstrating $2\times$ better performance on the

	Execution Cycles					**Timing**	**Area**	
Algorithm	small0	small1	small2	small3	large	ns	FFs	LUTs
(a)	116	203	179	165	23388	3.89	196	293
(b)	668	755	731	717	67267	4.62	2375	2157
(c)	63	157	128	114	11695	3.89	131	325

Table 5.1: Performance of Synthesized GCD Implementations – Performance characteristics of RTL generated from the GCD implementations shown in Figure 5.2. Performance for five input data sets is measured in executed cycles; area of the FPGA mapped design is shown in terms of flip-flops (FFs) and look-up tables (LUTs) used. Different implementations show a huge variance in the performance and physical characteristics depending on the complexity of the design synthesized.

large input dataset. This is due to the fact that the synthesized RTL for algorithm (a) includes an extra state in the controller finite-state machine which results in two cycles for each iteration of the algorithm rather than the one needed by (b).

A hand-written RTL implementation of (a) could optimize the extra cycle of latency away, making algorithms (a) and (b) more comparable. This highlights one current drawbacks of HLS: synthesis algorithms cannot always generate optimal implementations of a given algorithm without programmer help provided in the form of annotations. Future work for the HLSynthTool is adding the ability to insert annotations into PyMTL FL models in order to provide synthesis algorithms with these optimization hints.

5.2 Completing the Y-Chart: Physical Design in PyMTL

While many computer architects have no need to work at modeling abstractions lower than the register-transfer-level, there are an equal number of "VLSI architects" who are interested in exploring the system-level impact of physical-level optimizations. These researchers often forego automated synthesis and place-and-route tools for explicit specification of gate-level (GL) logic and manual, fine-grained placement of physical blocks. Use cases for these capabilities and techniques for enabling them in PyMTL are described below.

5.2.1 Gate-Level (GL) Modeling

RTL descriptions can be further refined to gate-level descriptions where each leaf model is equivalent to a simple boolean gate (e.g., standard cells, datapath cells, memory bit cell). For many

```
1  class OneBitFullAdderGL( Model ):         1  class OneBitFullAdderGLStruct( Model ):
2    def __init__( s ):                      2    def __init__( s ):
3                                             3
4      s.in0  = InPort ( 1 )                  4      s.in0  = InPort ( 1 )
5      s.in1  = InPort ( 1 )                  5      s.in1  = InPort ( 1 )
6      s.cin  = InPort ( 1 )                  6      s.cin  = InPort ( 1 )
7      s.sum  = OutPort( 1 )                  7      s.sum  = OutPort( 1 )
8      s.cout = OutPort( 1 )                  8      s.cout = OutPort( 1 )
9                                             9
10     @s.combinational                       10     s.xors = [Xor(1) for _ in range(2)]
11     def logic():                           11
12       a = s.in0                            12     s.connect_pairs(
13       b = s.in1                            13       s.in0,        s.xors[0].in[0],
14       c = s.cin                            14       s.in1,        s.xors[0].in[1],
15                                             15       s.xors[0].out, s.xors[1].in[0],
16       s.sum.value  = ( a ^ b ) ^ c         16       s.cin,        s.xors[1].in[1],
17       s.cout.value = ( a & b ) |           17       s.xors[1].out, s.sum,
18                      ( a & c ) |           18     )
19                      ( b & c )             19
                                              20     s.ands = [And(1) for _ in range(3)]
                                              21     s.ors  = [Or (1) for _ in range(2)]
                                              22
                                              23     s.connect_pairs(
                                              24       s.in0,        s.ands[0].in[0],
                                              25       s.in1,        s.ands[0].in[1],
                                              26       s.in0,        s.ands[1].in[0],
```

Figure 5.3: Gate-Level Modeling – Two
implementations of a gate-level, one-bit full-
adder: behaviorally using boolean equations
and structurally by instantiating gates and in-
terconnecting them. Behavioral are often more
concise as in the above example, but structural
approaches can be more parameterizable and
may be a more natural fit for extremely regular
structures.

```
                                              27       s.cin,        s.ands[1].in[1],
                                              28       s.in1,        s.ands[2].in[0],
                                              29       s.cin,        s.ands[2].in[1],
                                              30
                                              31       s.ands[0].out, s.ors [0].in[0],
                                              32       s.ands[1].out, s.ors [0].in[1],
                                              33       s.ors [0].out, s.ors [1].in[0],
                                              34       s.ands[2].out, s.ors [1].in[1],
                                              35     )
```

ASIC designers, this refinement step is usually handled automatically by synthesis and place-and-
route tools. However, adding support for gate-level models within PyMTL enables designers to
create parameterized cell tilers which can be useful when implementing optimized datapaths and
memory arrays. The functionality of these gate-level designs can potentially be implemented via
structural composition or by writing behavioral logic in concurrent blocks.

Figure 5.3 illustrates behavioral and structural implementations of a single-bit full-adder. Gate-
level behavioral representations use simple single-bit operators to implement boolean logic equa-
tions. In this implementation temporary variables are used to significantly shorten the boolean
equations. Gate-level structural representations instantiate and connect simple boolean gates. This

structural representation is considerably more verbose than the equivalent behavioral implementation, however, in cases where highly-parameterizable designs are needed, PyMTL's powerful structural elaboration cannot be emulated by behavioral logic.

The slice notation of PyMTL module lists, port lists, and bit-slices are particularly powerful when constructing parameterized bit-sliced datapaths. For example, composing an *n*-bit ripple-carry adder from *n* one-bit full-adders would use for loops and slicing to connect the *n*-bit inputs and outputs of the *n*-bit adder with the one-bit inputs and outputs of the each one-bit adder.

Gate-level modeling is supported naturally in PyMTL by the use of single-bit datatypes and boolean logic in behavioral logic. Structural gate-level modeling requires the creation of models for simple boolean gates such as ands, ors, and xors.

5.2.2 Physical Placement

The gate-level modeling described above is often used in tandem with manually implemented layout algorithms and physical placement. The use of these placement algorithms to introduce structured wiring in a custom or semi-custom design can greatly improve the power, performance, and area of an ASIC chip [DC00]. The PyMTL framework does not currently support these capabilities. This section describes an experimental `LayoutTool` that extracts placement information directly from PyMTL models annotated with parameterizable layout generators. The realization of this placement is shown in the form of SVG images output by the `LayoutTool`. In practice this geometry would be output into TCL files for consumption by an EDA tool.

The `LayoutTool` works by using introspection to first detect the existence of a `create_layout` method on elaborated models. If this method exists, the `LayoutTool` augments physical structures (e.g., submodules, wires) with placeholders for placement information and then executes the `create_layout` method. This method fills in these these placeholders with actual positioning information based on the layout algorithm and the size of submodules. Note that the `LayoutTool` and `create_layout` method take advantage of the signal, submodule, and parameter information already present on existing PyMTL models.

Figure 5.4 shows an example `create_layout` method for performing gate-level layout. Gate-level layout involves custom placement of individual cells, and can be particularly useful when developing parameterized array structures like arithmetic units, register files, and queues. The

```
1  class QueueGL( Model ):
2
3    def __init__( s, nbits, entries ):
4      s.regs    = [Reg( nbits ) for _ in range(entries)]
5      ...
6
7    def create_layout( s ):
8
9      entries = len( s.regs )
10     nbits   = s.regs[0].in_.nbits
11
12     for i in range( entries ):
13       for j in range( nbits ):
14         reg       = s.regs[ i * nbits + j ]
15         reg.dim.x = i * reg.dim.w
16         reg.dim.y = j * reg.dim.h
17
18     s.dim.w = entries * reg.dim.w
19     s.dim.h = nbits   * reg.dim.h
```
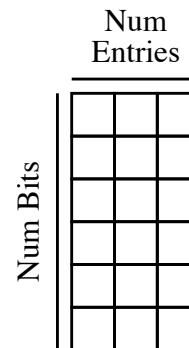
Figure 5.4: Gate-Level Physical Placement – Cell placement for one-bit register cells; creates an nbits-by-entries register array as shown in inset for use as the datapath in a small queue.

layout generator in Figure 5.4 creates parameterizable layout for register bit-cells which might be used to implement the datapath of a network queue.

Physical layout is also useful at coarser granularities than gate-level. Unit-level floorplanning, or micro-floorplanning, is useful for datapath tiling since datapaths often include structure that is destroyed by automated place-and-route tools. Figure 5.5 illustrates a structural implementation and simple datapath tiler for a *n*-ary butterfly router. Note that the tiler is able to leverage the component instances generated in the structural description during static elaboration. This unified approach simplifies building highly parameterized cross-domain designs.

At an even coarser granularity, macro-floorplans describe the placement of the top-level units. This is particularly important when determining the arrangement of components such as caches and processors in a multicore system. Figure 5.6 illustrates a floorplan generator for a simple ring network. The floorplan places routers in two rows and makes use of parameters passed to the parent (e.g., link_len) as well as parameters derived from the submodels (e.g., the router width/height). This enables parameterized floorplans which adapt to different configurations (e.g., larger routers, longer channels).

```python
from math import import log
class BFlyRouterDatapathRTL( Model ):
  def __init__( s, nary, nbits ):

    # ensure nary is > 1 and a power of 2
    assert nary > 1
    assert (nary & (nary - 1)) == 0

    sbits  = int( log( nary, 2 ) )
    N      = range( nary )

    s.in_ = [InPort (nbits)   for _ in N]
    s.val = [InPort (1     )  for _ in N]
    s.sel = [InPort (sbits)   for _ in N]
    s.out = [OutPort(nbits)   for _ in N]

    s.reg = [RegEn(nbits)       for _ in N]
    s.mux = [Mux(nary, nbits) for _ in N]

    for i in range( nary ):
      s.connect( s.in_[i],  s.reg[i].in_ )
      s.connect( s.val[i],  s.reg[i].en  )
      s.connect( s.sel[i],  s.mux[i].sel )
      s.connect( s.out[i],  s.mux[i].out )
      for j in range( nary ):
        s.connect( s.reg[i].out,
                   s.mux[j].in[i] )
```

```python
def create_layout( s ):
  offset = max( s.reg[0].dim.h,
                s.mux[0].dim.h )
  nary = len( s.reg )
  for i in range( nary ):
    s.reg[i].dim.x = 0
    s.mux[i].dim.x = s.reg[i].dim.w
    s.reg[i].dim.y = i * offset
    s.mux[i].dim.y = i * offset
  s.dim.w = reg[0].dim.w + mux[0].dim.w
  s.dim.h = offset * nary
```

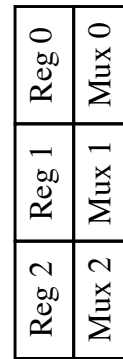| Reg 0	Mux 0
Reg 1	Mux 1
Reg 2	Mux 2

Figure 5.5: Micro-Floorplan Physical Placement – Naive tiling algorithm for *n*-bit register and *n*-bit mux models in a butterfly router datapath.

```python
class RingNetwork( Model ):
  ...
  def create_layout( s ):
    max = len( s.routers )
    for i, r in enumerate( s.routers ):
      if i < (max / 2):
        r.dim.x = i * ( r.dim.w + s.link_len )
        r.dim.y = 0
      else:
        r.dim.x = (max - i - 1) * ( r.dim.w + s.link_len )
        r.dim.y = r.dim.h + s.link_len

    s.dim.w = max/2 * r.dim.w + (max/2 - 1) * s.link_len
    s.dim.h = 2 * r.dim.h + s.link_len
```
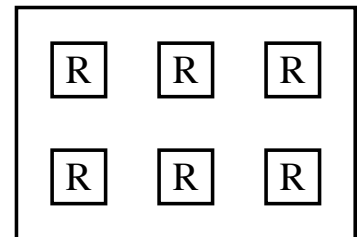

Figure 5.6: Macro-Floorplan Physical Placement – Floorplanning algorithm for a parameterizable ring network. Algorithm places routers in two rows as shown in the diagram. This example illustrates the hierarchical propagation of geometry information.

Figure 5.7: Example Network Floorplan – Post place-and-route chip plot for a 16 node network; floorplan was derived from the prototype PyMTL `LayoutTool` applied to a parameterized RTL model of the network. A more comprehensive physical model would include floorplanning for individual processors, memories, and routers as well.

A demonstration of a simple macro-floorplanner in action can be seen in Figure 5.7, which shows a post place-and-route chip plot of a sixteen node network. The top-level floorplan of this design was generated by applying the `LayoutTool` to an instantiated and elaborated PyMTL RTL model. The generated TCL file containing placement and geometry information and Verilog RTL for the network were then passed into a Synopsys EDA toolflow to synthesize, place, and route the design. A significant benefit of this approach is that floorplanning information and RTL logic is specified in a single location and parameterized using a unified mechanism. This ensures the physical floorplanning information stays synchronized with the behavioral RTL logic as parameters to the design are changed.

Physical placement is typically described in a significantly different environment from what is used RTL modeling, complicating the creation of a single unified and parameterizable design description. Using the `LayoutTool`, PyMTL allows physical layout to be described alongside the RTL specification of a given component. In addition to providing the ability to utilize information generated from static elaboration of the RTL model for physical layout, this configuration enables a hierarchical approach to layout specification: starting at the leaves of a design each model places their children, then uses this information to set their own dimensions. As a physical layout algorithm progresses up the hierarchy, placement information from lower-level models is used to inform layout decisions in higher-level models.

CHAPTER 6
CONCLUSION

The work presented in this thesis has aimed to address a critical need for future hardware system design: productive methodologies for exploring and implementing vertically integrated computer architectures. Possible solutions to the numerous challenges associated with vertically integrated computer architecture research were investigated via the construction of two prototype frameworks named PyMTL and Pydgin. Each of these frameworks proposed a novel design philosophy that combined embedded domain-specific languages (EDSLs) with just-in-time (JIT) optimization techniques. Used in tandem, EDSLs and JIT optimization can enable both improved designer productivity and high-performance simulation.

To conclude, the following sections will summarize the contents of this thesis, discuss its primary contributions, and end with a presentation of possible directions for future work.

6.1 Thesis Summary and Contributions

This thesis began by discussing abstractions used in hardware modeling and several taxonomies used to categorize models based on these abstractions. To address some of the limitations of these existing taxonomies, I proposed a new hardware modeling taxonomy that classifies models based on their behavioral, timing, and resource accuracy. Three model classes commonly used in computer architecture research — functional-level (FL), cycle-level (CL), and register-transfer-level (RTL) — were qualitatively introduced within the structure of this taxonomy. Computer research methodologies built around each of the FL, CL, and RTL model classes were discussed, before describing the challenges of the *methodology gap* that arises from the distinct modeling languages, patterns, and tools used by each of these methodologies. Several approaches to close this methodology gap were suggested, and a new vertically integrated research methodology called *modeling towards layout* was proposed as a way to enable productive, vertically integrated, computer architecture research for future systems.

The remainder of this thesis introduced several tools I constructed to enable the *modeling towards layout* research methodology and promote more productive hardware design in general. Each of these tools utilized a common design philosophy based on the construction of embedded domain-specific languages within Python. This design approach serves the dual purpose of (1) significantly reducing the design time for constructing novel hardware modeling frameworks and (2)

103

providing a familiar, productive modeling environment for end users. Two of these tools, PyMTL and Pydgin, additionally incorporate just-in-time (JIT) optimization techniques as a means to address the simulation performance disadvantages associated with implementing hardware models in a productivity-level language such as Python.

The major contributions of this thesis are summarized below:

- **The PyMTL Framework** – The core of PyMTL is a Python-based embedded-DSL for concurrent-structural modeling that allows users to productively describe hardware models at the FL, CL, and RTL levels of abstraction. The PyMTL framework exposes an API to these models that enables a model/tool split, which in turn facilitates extensibility and modular construction of tools such as simulators and translators. A provided simulation tool enables verification and performance analysis of user-defined PyMTL models, while additionally allowing the co-simulation of FL, CL and RTL models.

- **The PyMTL to Verilog Translation Tool** – The PyMTL Verilog translation tool gives users the ability to automatically convert their PyMTL RTL into Verilog HDL; this provides a path to EDA toolflows and enables the collection of accurate area, energy, and timing estimates. This tool leverages the open-source Verilator tool to compile generated Verilog into a Python-wrapped component for verification withhin existing PyMTL test harnesses and co-simulation with PyMTL FL and CL models. The Verilog translation tool is a considerable piece of engineering work that makes the modeling towards layout vision possible, and is also the target of ongoing research in providing higher-level design abstractions that are Verilog-translatable.

- **The SimJIT Specializer** – SimJIT performs just-in-time specialization of PyMTL CL and RTL models into optimized, high-performance C++ implementations, greatly improving the performance of PyMTL simulations. While SimJIT-CL was only a proof-of-concept that worked on a very limited subset of models, this prototype showed that this is a promising approach for achieving high-performance simulators from high-level language specifications. SimJIT-RTL is a mature specializer that takes advantage of the open-source Verilator tool, and is in active use in research.

- **The Pydgin Framework** – Pydgin implements a Python-based, embedded architecture description language that can be used to concisely describe ISAs. These embedded-ADL de-

scriptions provide a productive interface to the RPython translation toolchain, an open-source framework for constructing JIT-enabled dynamic language interpreters, and creatively repurposes this toolchain to produce instruction set simulators with dynamic binary translation. The Pydgin framework also provides numerous library components to ease the process of implementing new ISSs. Pydgin separates the process of specifying new ISAs from the performance optimizations needed to create fast ISSs, enabling cleaner implementations and simpler verification.

- **ISS-Specific Tracing-JIT Optimizations** – In the process of constructing Pydgin, significant analysis was performed to determine which JIT optimizations improve the performance of Pydgin-generated instruction set simulators. Several optimizations were found that were unique to ISSs; this thesis described these various optimizations and characterized their performance impact on Pydgin ISSs.

- **Practical Pydgin ISS Implementations** – In addition to creating the Pydgin framework itself, three ISSs were constructed to help evaluate the productivity and performance of Pydgin. The SMIPS ISA is used internally by my research group for teaching and research; the Pydgin SMIPS implementation handily outperformed our existing ISS and has now been adopted as an actively used replacement. The ARMv5 ISA is widely used in the greater architecture research and hardware design community. My hope is that these communities will build upon our initial partial implementation in Pydgin. There has already been some recent activity in this direction. The RISC-V ISA is quickly gaining traction in both academia and industry. Like the Pydgin ARMv5 implementation, my hope is that the Pydgin RISC-V ISS will find use in the computer architecture research community. Outside researchers have begun using Pydgin to build simulators for other ISAs. This is especially encouraging, and I hope that this trend continues.

- **Extensions to the Scope of Design in PyMTL** – Chapter 5 introduced two experimental tools for expanding the scope of hardware design in PyMTL. The first tool, `HLSynthTool` provides a very basic demonstration of high-level synthesis from PyMTL FL models. Although this tool is just an initial proof-of-concept, it proves that PyMTL FL models can undergo automated synthesis to Verilog RTL, and this generated RTL can be wrapped and tested within the PyMTL framework. The second tool, `LayoutTool` demonstrates how PyMTL

models can be annotated with physical placement information, enabling the creation of powerful, parameterizable layout generators. This helps close the methodology gap between logical design and physical design, and better couples gate-level logic design and layout.

6.2 Future Work

The work in this thesis serves as a launching point for many possible future directions in constructing productive hardware design methodologies. A few areas of future work are listed below:

- **More Productive Modeling Abstractions in PyMTL** – PyMTL has added a number of library components key to improving the productivity of designers in PyMTL. These components range from simple encapsulation classes that simplify the organization and accessing of data (e.g., `PortBundles`, `BitStructs`), to complex proxies that abstract away latency-insensitive interfaces with user-friendly syntax (e.g., `QueueAdapters`, `ListAdapters`). However, these examples only scratch the surface of possibilities for improving higher-level modeling abstractions for hardware designers. Just a few ideas include: embedded-DSL extensions for specifying state machines and control-signal tables, Verilog-translatable method-based interfaces, alternative RTL specification modes such as guarded atomic actions or Chisel-like generators, embedded-ADLs for processor specification, and special datatypes for floating-point and fixed-point computations.

- **PyMTL Simulator Performance Optimizations** – SimJIT was an initial attempt at addressing the considerable performance challenges of using productivity-level languages for hardware modeling. The prototype implementation of SimJIT-CL only generated vanilla C++ code and did not explore advanced performance optimizations. Two specific opportunities for improving simulator performance include generating parallel simulators for PyMTL models and replacing the combinational logic event queue with a linearized execution schedule. The concurrent execution semantics of `@tick` annotated blocks and the double-buffering of next signals provide the opportunity for all sequential blocks to be executed in parallel; however, determining the granularity of parallel execution and co-scheduling of blocks are open areas of research. Execution of combinational blocks is implemented using sensitivity lists and a dynamic event queue, this could alternatively be done by statically determining

an execution schedule based on signal dependencies. Although this approach would add an additional burden on users to not create blocks with false combinational loops, it could create opportunities for inter-block statement reordering and logic optimization.

- **Improvements to the Pydgin Embedded-ADL** – The embedded-ADL in Pydgin provides a concise way to specify the semantics for most instruction operations, however, there are a number of areas for improvement. Currently, no bit-slicing syntax exists for instructions and instruction fields requiring verbose shifting and masks to access particular bits. Another challenge is communicating bit-widths to the RPython translation toolchain: RPython provides datatypes for specifying which C datatypes a value should fit into. This is not as fine-grained or as clean as it should be. Finally, specifying the semantics of floating-point instructions is rather difficult to do in Pydgin, and the addition of a specific floating-point datatype could help considerably.

- **Toolchain Generation from the Pydgin Embedded-ADL** – Perhaps the most difficult task when creating a Pydgin ISS for a new ISA is the development of the corresponding cross-compiler and binutils toolchain. Prior work has leveraged ADLs as a specification language not only for instruction set simulators, but for automatically generating toolchains as well. The addition of such a capability to Pydgin would considerably accelerate the process of implementing new ISAs, and greatly ease the burden of adding new instructions to existing ISAs. A primary challenge is determining how to encode microarchitectural details of importance to the compiler without introducing significant additional complexity to the ADL.

- **Pydgin JIT Performance Optimizations** – One significant area of potential improvement for Pydgin DBT-ISSs is reducing the performance variability from application to application. While Pydgin's tracing-JIT performs exceptionally well on some benchmarks, it occasionally achieves worse performance than page-based JITs. Improving the performance of the tracing-JIT on highly-irregular applications, reducing the JIT warm-up time, and adding explicit RPython optimizations for bit-specific types are several opportunities for future work.

- **PyMTL-based High-Level Synthesis** – Another promising avenue for improving the productivity of RTL designers is high-level synthesis (HLS), which can take untimed or partially timed algorithm specifications as input and directly generate Verilog HDL. There are a number of potential advantages of using PyMTL as an input language to HLS rather than C or

C++, particularly with respect to integration challenges associated with verifying and co-simulating generated RTL. The experimental `HLSynthTool` introduced in this thesis is only an initial step towards a full PyMTL HLS solution. An ideal PyMTL-based HLS flow would be able to automatically translate PyMTL FL models directly into PyMTL (rather than Verilog) RTL, and perform this synthesis without relying on commerical 3rd-party synthesis tools. Three possible areas for future work on PyMTL-based HLS include the following: conversion of PyMTL FL models into LLVM IR, conversion of IR into PyMTL RTL, and a pure Python tool for scheduling and logic optimization of hardware-aware IR.

- **Productive Physical Design in PyMTL** – PyMTL is designed to provide a productive environment for FL, CL, and RTL modeling, however, PyMTL does not currently provide support for the physical design processes used in my ASIC design flows. Section 5 discussed some initial experiments into implementing parameterizable and context-dependent floorplanners using PyMTL. However, there are many more opportunities for facilitating productive physical design with PyMTL, including memory compilers, verification tools, design rule checkers, and specification of crafted datapath cells.

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.

[ALE02]   T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, Feb 2002.

[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.

[BBB$^+$11]   N. Binkert, B. 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*, 39(2):1–7, Aug 2011.

[BCF$^+$11]   C. F. Bolz, A. Cuni, M. Fijałkowski, 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.

[BCFR09]   C. F. Bolz, A. Cuni, M. Fijałkowski, 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.

[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.

[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.

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

[BG04]   M. Burtscher and I. Ganusov. Automatic Synthesis of High-Speed Processor Simulators. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2004.

[BGOS12]   A. Butko, R. Garibotti, L. Ost, and G. Sassatelli. Accuracy Evaluation of GEM5 Simulator System. *Int'l Workshop on Reconfigurable Communication-Centric Systems-on-Chip*, Jul 2012.

[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.

[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.

[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.

[BMGA05]   B. Bailey, G. Martin, M. Grant, and T. Anderson. *Taxonomies for the Development and Verification of Digital Systems*. Springer, 2005. URL http://books.google.com/books?id=i04n_I4EOMAC

[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*, 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.

[BTM00]   D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2000.

[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.

[BXF13]   P. Birsinger, R. Xia, and A. Fox. Scalable Bootstrapping for Python. *Int'l Conf. on Information and Knowledge Management*, Oct 2013.

[CBHV10] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing MATLAB through Just-In-Time Specialization. *Int'l Conf. on Compiler Construction*, Mar 2010.

[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.

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

[CKL+09] B. Catanzaro, S. Kamiland, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization. *Workshop on Programming Models for Emerging Architectures*, Sep 2009.

[CLSL02] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti. Precise and Accurate Processor Simulation. *Workshop on Computer Architecture Evaluation Using Commercial Workloads*, Feb 2002.

[CMSV01] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Sep 2001.

[CNG+06] J. C. Chaves, J. Nehrbass, B. Guilfoos, J. Gardiner, S. Ahalt, A. Krishnamurthy, J. Unpingco, A. Chalker, A. Warnock, and S. Samsi. Octave and Python: High-level Scripting Languages Productivity and Performance Evaluation. *HPCMP Users Group Conference*, Jun 2006.

[DBK01] R. Desikan, D. Burger, and S. W. Keckler. Measuring Experimental Error in Microprocessor Simulation. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2001.

[DC00] W. J. Dally and A. Chang. The Role of Custom Design in ASIC Chips. *Design Automation Conf. (DAC)*, Jun 2000.

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

[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.

[EH92] W. Ecker and M. Hofmeister. The Design Cube - A Model for VHDL Designflow Representation. *EURO-DAC*, pages 752–757, Sep 1992.

[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.

[FM11] S. H. Fuller and L. I. Millet. Computing Performance: Game Over or Next Level? *IEEE Computer*, 44(1):31–38, Jan 2011.

[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.

[GK83] D. D. Gajski and R. H. Kuhn. New VLSI Tools. *IEEE Computer*, 16(12):11–14, Dec 1983.

[GKB09] M. Govindan, S. W. Keckler, and D. Burger. End-to-End Validation of Architectural Power Models. *Int'l Symp. on Low-Power Electronics and Design (ISLPED)*, Aug 2009.

[GKO+00] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich. FLASH vs. (Simulated) FLASH: Closing the Simulation Loop. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Dec 2000.

[GPD+14] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, N. Paver, and N. K. Jha. Sources of Error in Full-System Simulation. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar 2014.

[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.

[HA03] J. C. Hoe and Arvind. Operation-Centric Hardware Description and Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2003.

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

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

[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.

[Hud96] P. Hudak. Building Domain-Specific Embedded Languages. *ACM Comupting Surveys*, 28(4es), Dec 1996.

[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.

[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.

[KLPS09]   A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi. ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. *Design, Automation, and Test in Europe (DATE)*, Apr 2009.

[KLPS11]   A. B. Khang, B. Li, L.-S. Peh, and K. Samadi. ORION 2.0: A Power-Area Simulator for Interconnection Networks. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, PP(99):1–5, Mar 2011.

[LAS+13]   S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing. *ACM Trans. on Architecture and Code Optimization (TACO)*, 10(1), Apr 2013.

[LB14]   D. Lockhart and C. Batten. Hardware Generation Languages as a Foundation for Credible, Reproducible, and Productive Research Methodologies. *Workshop on Reproducible Research Methodologies (REPRODUCE)*, Feb 2014.

[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.

[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.

[LM10]   E. Logaras and E. S. Manolakos. SysPy: Using Python For Processor-centric SoC Design. *Int'l Conf. on Electronics, Circuits, and Systems (ICECS)*, Dec 2010.

[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.

[Mad95]   V. K. Madisetti. System-Level Synthesis and Simulation in VHDL – A Taxonomy and Proposal Towards Standardization. *VHDL International Users Forum*, Spring 1995.

[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.

[MBJ09]   N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches, 2009.

[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.

[Nik04]   N. Nikhil. Bluespec System Verilog: Efficient, Correct RTL from High-Level Specifications. *Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, Jun 2004.

[num14]   Numba. Online Webpage, accessed Oct 1, 2014.
           http://numba.pydata.org.

[Oli07]   T. E. Oliphant. Python for Scientific Computing. *Computing in Science Engineering*, 9(3):10–20, 2007.

[PA03]   D. A. Penry and D. I. August. Optimizations for a Simulator Construction System Supporting Reusable Components. *Design Automation Conf. (DAC)*, Jun 2003.

[Pan01]   P. R. Panda. SystemC: A Modeling Platform Supporting Multiple Design Abstractions. *Int'l Symp. on Systems Synthesis*, Oct 2001.

[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.

[Pen06]   D. A. Penry. *The Acceleration of Structural Microarchitectural Simulation Via Scheduling*. Ph.D. Thesis, CS Department, Princeton University, Nov 2006.

[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 (IS-PASS)*, Apr 2011.

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

[PFH+06]   D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors. Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multi-processors. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2006.

[PHM00]   S. Pees, A. Hoffmann, and H. Meyr. Retargetable Compiled Simulation of Embedded Processors Using a Machine Description Language. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, Oct 2000.

[Piz14]   F. Pizio. Introducing the WebKit FTL JIT. Online Article (accessed Sep 26, 2014), May 2014. https://www.webkit.org/blog/3362/introducing-the-webkit-ftl-jit.

[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.

[Pre00]   L. Prechelt. An Empirical Comparison of Seven Programming Languages. *IEEE Computer*, 33(10):23–29, Oct 2000.

[pyp11]   PyPy. Online Webpage, 2011 (accessed Dec 7, 2011).
            `http://www.pypy.org`.

[pyt14a]   PyTest. Online Webpage, accessed Oct 1, 2014).
            `http://www.pytest.org`.

[pyt14b]   PyTest Coverage Reporting Plugin. Online Webpage, accessed Oct 1, 2014. `https://pypi.python.org/pypi/pytest-cov`.

[pyt14c]   PyTest Distributed Testing Plugin. Online Webpage, accessed Oct 1, 2014. `https://pypi.python.org/pypi/pytest-xdist`.

[pyt14d]   Greenlet Concurrent Programming Package. Online Webpage, accessed Oct 1, 2014. `http://greenlet.readthedocs.org`.

[QDZ06]   W. Qin, J. D'Errico, and X. Zhu. A Multiprocessing Approach to Accelerate Retargetable and Portable Dynamic-Compiled Instruction-Set Simulation. *Int'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.

[RBMD03]   M. Reshadi, N. Bansal, P. Mishra, and N. Dutt. An Efficient Retargetable Framework for Instruction-Set Simulation. *Int'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.

[RHWS12]   A. Rubinsteyn, E. Hielscher, N. Weinman, and D. Shasha. Parakeet: A Just-In-Time Parallel Accelerator for Python. *USENIX Workshop on Hot Topics in Parallelism*, Jun 2012.

[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.

[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.

[SCK⁺12]   C. Sun, C.-H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic. DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling. *Int'l Symp. on Networks-on-Chip (NOCS)*, May 2012.

[SCV06]   P. Schaumont, D. Ching, and I. Verbauwhede. An Interactive Codesign Environment for Domain-Specific Coprocessors. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 11(1):70–87, Jan 2006.

[SIT⁺14]   S. Srinath, B. Ilbeyi, M. Tan, G. Liu, Z. Zhang, and C. Batten. Architectural Specialization for Inter-Iteration Loop Dependence Patterns. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[SRWB14]   Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks. Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2014.

[SWD⁺12]   O. Shacham, M. Wachs, A. Danowitz, S. Galal, J. Brunhaver, W. Qadeer, S. Sankaranarayanan, A. Vassilev, S. Richardson, and M. Horowitz. Avoiding Game Over: Bringing Design to the Next Level. *Design Automation Conf. (DAC)*, Jun 2012.

[sys14]   SystemC TLM (Transaction-level Modeling). Online Webpage, accessed Oct 1, 2014. http://www.accellera.org/downloads/standards/systemc/tlm.

[Tho13]   E. W. Thomassen. *Trace-Based Just-In-Time Compiler for Haskell with RPython*. M.S. Thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, 2013.

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

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

[Tra05]   L. Tratt. Compile-Time Meta-Programming in a Dynamically Typed OO Language. *Dynamic Languages Symposium (DLS)*, Oct 2005.

[ver13]   Verilator. Online Webpage, 2013 (accessed July, 2013). http://www.veripool.org/wiki/verilator.

[VJB+11]   J. I. Villar, J. Juan, M. J. Bellido, J. Viejo, D. Guerrero, and J. Decaluwe. Python as a Hardware Description Language: A Case Study. *Southern Conf. on Programmable Logic*, Apr 2011.

[vPM96]   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.

[VVA04]   M. Vachharajani, N. Vachharajani, and D. I. August. The Liberty Structural Specification Language: A High-Level Modeling Language for Component Reuse. *ACM SIG-PLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun 2004.

[VVP+02]   M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural Exploration with Liberty. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2002.

[VVP+06]   M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August. The Liberty Simulation Environment: A Deliberate Approach to High-Level System Modeling. *ACM Trans. on Computer Systems (TOCS)*, 24(3):211–249, Aug 2006.

[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.

[WPM02]   H. Wang, L.-S. Peh, and S. Malik. Orion: A Power-Performance Simulator for Interconnection Networks. *Int'l Symp. on Microarchitecture (MICRO)*, Nov 2002.

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

[ZHCT09]   M. Zhang, G. Hu, Z. Chai, and S. Tu. Trilobite: A Natural Modeling Framework for Processor Design Automation System. *Int'l Conf. on ASIC*, Oct 2009.

[ZTC08]   M. Zhang, S. Tu, and Z. Chai. PDSDL: A Dynamic System Description Language. *Int'l SoC Design Conf.*, Nov 2008.