PRODUCTIVE AND EXTENSIBLE HARDWARE MODELING, SIMULATION, AND VERIFICATION METHODOLOGIES

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
Shunning Jiang
August 2021

PRODUCTIVE AND EXTENSIBLE HARDWARE MODELING, SIMULATION, AND VERIFICATION METHODOLOGIES

Shunning Jiang, Ph.D.
Cornell University 2021

As Dennard scaling broke down in the 2000s and Moore's Law slowed down in the 2010s, computer engineers have been exploring new ways to extract more computing performance without increasing the power density or the transistor count. Various specialized hardware accelerators are integrated into existing multi-core architectures, creating heterogeneous system-on-chips (SoC). However, as more heterogeneous SoCs are built, the number of different hardware blocks in a single SoC is rapidly increasing. This trend significantly increases the non-recurring engineering (NRE) cost required to build new SoCs. Maximizing the reuse of hardware blocks across and inside SoC designs is one of the key ways to reduce the NRE cost. This requires both flexible parameterization of a single hardware design block and versatile composition of numerous different hardware design blocks. To enable and maximize such reuse of hardware blocks, productive hardware modeling methodologies play a critical role in the modern computer engineering workflow.

This thesis takes an engineering research approach to explore productive and extensible hardware modeling, simulation, and verification methodologies. I identify four major challenges in state-of-the-art productive hardware modeling methodologies and formulate each challenge into a stand-alone research question. Then, I propose several techniques to address these research questions: (1) native in-memory intermediate representation (NIMIR), a novel modular framework architecture, to improve the flexibility and extensibility of hardware generation and simulation frameworks (HGSF); (2) unified modular ordering constraints (UMOC), a novel modeling technique coupled with scheduling algorithms, to unify cycle- and register-transfer-level modeling and achieve high model fidelity with little effort; (3) Mamba++, a series of HGSF-aware just-in-time compilation (JIT) techniques and JIT-aware HGSF design techniques, to close the simulation performance gap in HGSFs; and (4) PyH2, our vision and techniques for testing various hardware designs leveraging open-source software, to reduce testing/verification time for agile hardware de-

sign flows. Finally, in addition to addressing each individual research question, I created PyMTL3, a new hardware generation and simulation framework which incorporates the techniques proposed in this thesis. By implementing the techniques inside a real hardware modeling framework, the practicality of the proposed techniques is demonstrated. PyMTL3 has been used in courses at Cornell University, in various research projects, and in several advanced-node chip tape-outs.

BIOGRAPHICAL SKETCH

Shunning Jiang was born on April 18, 1993 to Dahuo Jiang and Zhijin Chen in Shaoxing, Zhejiang, China. At a young age, he found himself not only interested in but also capable of computing and programming. He started casually writing some BASIC and Pascal programs at Shaoxing Beihai Elementary School, participated in National Olympiad in Informatics in Province from junior group to senior group at Shaoxing No.1 Junior Middle School and Shaoxing No.1 High School, and was very fortunate to have a chance to participate in the National Olympiad in Informatics. During these years he discovered that he was slightly more interested in computer engineering related fields than computer science related fields.

Shunning was accepted to Shanghai Jiaotong University as an undergraduate student after he was the highest ranked Bronze medal finalist (oops) in National Olympiad in Informatics. He attended the undergraduate program (ACM Honored Class/Computer Science in Zhiyuan College) and met Shuang Chen who became his wife four years later. He struggled with various hardcore math curriculums in the first few years, but enjoyed courses and projects in compiler, computer architecture, operating system, and database. He spent a lot of time dating his girlfriend in the library and tried really hard to make academic progress. He made up his mind to pursuit a doctoral degree in the US after trying out some research projects in Advanced Computer Architecture Laboratory at SJTU and Xtra Computing Group at Nanyang Technological University.

Shunning decided to join Cornell University as a Ph.D. student. He started to work with Professor Christopher Batten after he realized he liked Prof. Batten's perspective in computer engineering research. He picked up a few projects, but he chose to become an all-around computer engineer specialized in hardware modeling methology. He did one internship at Google right before his daughter was born, where he worked on automatically scheduling Halide image-processing pipelines. He believes the whole Ph.D. journey at Cornell University was very worthwhile. He sometimes wishes the last one and half year of his Ph.D. career had not been affected by the COVID-19 global pandemic.

This document is dedicated to my parents, my beloved wife Shuang Chen,
and my daughter Carly Xin Jiang.

ACKNOWLEDGEMENTS

My graduate career is totally different from what I imagined before coming to Ithaca. I am really grateful to those who have supported me throughout this journey.

First of all, I would like to thank my advisor Christopher Batten. I vividly remember many moments in the last six years. At the very beginning, Chris told me that I needed to improve my English speaking skills before joining the Batten Research Group, otherwise he would not be able to communicate with me efficiently. In one semester, his phone call at 5:10pm every day during his walk home always brought disruptive and ground-breaking ideas to my ongoing work. During the second summer, Chris and I met every single afternoon to push the research progress as fast as we could. After Carly was born in my fourth year, Chris always reminded me to balance work and life. During two months in my fifth year, Chris, Yanghui, Peitian, and I submitted five papers, skied with Princeton folks, and traveled to a DARPA meeting in Salt Lake City. There were countless brainstorming session in Rhodes Hall, over the beam robot, and of course, over Zoom during the COVID-19 months. Chris made me understand the importance of open and honest communications, the importance of asking any question whether it is stupid or not, the importance of teaching, etc. I also want to thank the rest of my thesis committee, Prof. José Martínez and Prof. Christina Delimitrou, for their guidance, feedback, and support for me and my family along the way. José usually provides different but useful perspectives to my questions. Christina is always encouraging and supportive.

I would like to thank members of the Batten Research Group for guidance and collaboration. I am thankful to Ji Kim and Shreesha Srinath for sharing their wisdoms in computer architecture during my junior years. I was very fortunate to have Christopher Torng as a role model for more than half of my six years in BRG, influencing me with his working ethics, research methodology, and ways of thinking. I was also very fortunate to collaborate with the wizard hacker Berkin Ilbeyi on various projects, where I was always surprised at how fast Berkin came up with a cool solution. Moyang Wang was a really good friend to share those ups and downs. Khalid Al-Hawaj, I have learnt a lot from you, and I hope your knowledge continues to grow indefinitely. Tuan Ta, it was great fun to work with you on the BRG-I2OL processor project and I really enjoyed those cheerful daily conversations with you. Lin Cheng, you have become the next-gen wizard hacker in BRG. The progression and inheritance in BRG was pretty magical, ha! In 2017, I was getting help on hacking PyPy from Berkin when Lin was not even in BRG. Then in 2020 I was getting

help from Lin. I would like to thank Yanghui Ou and Peitian Pan for working with me in those key components of my thesis. Without you guys I would have to spend more time working on those projects alone. I also hope I had good influence on you two. Nick Cebry, keep up the great work you have been doing. Also as a member of the Computer Systems Laboratory, I would like to thank all my friends at CSL. Thanks Yuan Zhou, Weizhe Hua, Yu Gan and Yanqi Zhang for many things in work and life. I also want to thank Ritchie Zhao, Steve Dai, Hanchen Jin, Sachille Atapattu, Nitish Srivastava, Helena Caminal, and Mark Buckler for their support. Thanks Prof. Zhiru Zhang and Prof. Adrian Sampson for advices and feedbacks.

I owe so much to all the people who nurtured me along the way from a kid who liked to hack computers to an experienced researcher/engineer with a doctor of philosophy in computer engineering. I remember the days when I sat in the middle school computer room learning from Ms. Sijie Wang. Then Ms. Heli Chen and Mr. Hongxiang Shao gave me a chance to learn more about algorithms and competitive programming in high school. Prof. Yong Yu brought me into the prestigious ACM honored class undergraduate program and part of Zhiyuan College. Prof. Xiaoyao Liang and Prof. Naifeng Jing provided me with immersive experience of computer architecture research in my junior year. I want to thank Prof. John Hopcroft for bringing the whole batch of students to Cornell in the summer right before my senior year, where I was fortunate enough to meet Mr. (now Dr.) Xiaodong Wang in a party held by Prof. David Gries. I want to thank Prof. Bingsheng He and Prof. Xueyan Tang me how to write a research paper in Singapore. Thanks to Dr. Jing Pu for kindly hosting me at Google for an inspirational internship.

Finally, I would like to thank my wife Shuang Chen for literally *everything* in the last ten years. There are simply no words that can describe how much you mean to me. Carly Jiang, you are the silly little girl who has changed my life. This thesis would not be possible without my parents and my parents-in-law who came to a different country to help take care of Carly so that I am able to work in the office as usual. I also want to thank Ithaca Community Childcare Center (IC3) for providing Carly a COVID-free environment during weekdays in the last year.

In terms of funding, this thesis was supported in part by Cornell Graduate School Fellowship, Richard E. Lunquist Graduate Award, NSF SHF Award #1527065, NSF CRI Award #1512937, AFOSR YIP Award #FA9550-15-1-0194, DARPA SDH Award #FA8650-18-2-7863, DARPA POSH Award #FA8650-18-2-7852, DARPA CRAFT Award #HR0011-16-C-0037, a research gift from Xilinx, Inc., and the the Center for Applications Driving Architectures (ADA), one of six centers

of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA. This work was also supported by equipment, tool, and/or physical IP donations from Intel, Xilinx, Synopsys, Cadence, and ARM. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation theron. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

SoC	systems-on-chip
NRE	non-recurring engineering
HDL	hardware description language
HPF	general-purpose graphics processing unit
HGF	single-instruction multiple-data
HGSF	single-instruction multiple-thread
JIT	reduced instruction set computer
API	application programming interface
DSL	domain-specific language
DUT	design under test
TB	test bench
FPGA	field-programmable gate array
ASIC	application-specific integrated circuit
RTL	register-transfer level
CL	cycle level
FL	functional level
IR	intermediate representation
IP	intellectual property
EDA	electronic design automation
NIMIR	native in-memory intermediate representation
RTLIR	register-transfer level intermediate representation
UMOC	unified modular ordering constraints
HLS	high-level synthesis
AST	abstract syntax tree
VCD	value change dump
SCC	strongly connected components
TLM	transactional-level modeling
UDG	unified directed graph
DAG	directed acyclic graph
MDU	multiply/divide unit
FPU	floating point unit
HSS	hierarchical static scheduling
AOT	ahead-of-time
TLB	translation lookaside buffer
UVM	universal verification methodology
CRT	complete-random testing
IDT	iterative-deepened testing
PBT	property-based testing
ISA	instruction set architecture

CHAPTER 1
INTRODUCTION

The twentieth century witnessed almost exponential growth in *single-core* computing performance thanks to Dennard scaling [DGY+74] and Moore's Law [Moo65]. However, in the 2000s, Dennard scaling broke down due to the increasing power density and heat dissipation, which drastically increased the complexity to extract more single-core performance. To fully utilize the increasing transistor count without enlarging the power envelope, the mainstream computing platforms raced towards *multi-core and multi-processor architectures* [KFJ+03, KTR+04, EBA+11] running various parallel applications [ope08, Rei07, MRR12, ARKK13]. Then, in the 2010s, the slowdown of Moore's Law delayed the delivery of new technology nodes with higher transistor density. As a result, computer engineers have been radically exploring ways to extract more computing performance without increasing the transistor count. *Hardware specialization*, an approach to trade off flexibility for performance and/or energy-efficiency, quickly becomes an appealing option. Various specialized hardware accelerators are integrated into existing multi-core architectures, which becomes a new type of computing platform called *heterogeneous system-on-chip (SoC)* [WJM08, Tay13]. As of today, heterogeneous SoCs can be found in almost all contemporary computing devices. State-of-the-art computing chips [KJJ+20, VSS+20, PMH+21] usually include: (1) asymmetric multi-core processors such as a mix of out-of-order cores, in-order cores, and cores with different frequency domains [Gre11, LK09]; (2) various domain-specific programmable architecture such as general-purpose graphics processing units [KDK+11], programmable manycore accelerators [MFN+17, RZAH+19, KJT+17, Bol12, SGC+16, BCC+17], and coarse-grained reconfigurable arrays [PFKM06, PZK+17, GHN+12]; and/or (3) many highly specialized accelerators such video/audio codecs, neural network accelerators [CKES17], and data encryption/decryption engines.

However, as computer engineers build more heterogeneous SoCs, the number of different hardware blocks in a single SoC is rapidly increasing. This trend leads to significant increasing non-recurring engineering (NRE) costs of building new SoCs [SWD+12]. Maximizing the reuse of hardware blocks across/inside SoC designs is one of the key ways to reduce the NRE cost, which requires both flexible parameterization of a single hardware design block and versatile composition of numerous different hardware design blocks. To enable and maximize such reuse of hardware

blocks, productive hardware modeling methodologies play a critical role in the modern computer engineering workflow.

This thesis proposes new techniques to enable state-of-the-art hardware modeling methodologies to better reduce NRE costs in heterogeneous SoCs. This thesis also presents a new open-source hardware modeling framework that incorporates these new techniques.

1.1 State-of-the-Art Hardware Modeling Methodologies

Computer engineers have been combating against the high NRE costs caused by the parameterization and composition challenges from heterogeneous SoC design. Developing a pool of highly parametrized and thoroughly tested hardware "generators" is a compelling solution to increase the reuse of hardware blocks across different chips or even inside the same chip. Several generations of productive hardware modeling frameworks with different workflows have been built to effectively architect, build, verify, and maintain highly parametrized RTL blocks.

Hardware Description Languages (HDL) – Probably the most prevalent approach of building hardware is to write register-transfer level (RTL) descriptions using HDLs (e.g., VHDL [Ped20], Verilog [TM08]). HDLs were originally introduced in the 1970s to accommodate the explosion of the number of transistors in a chip by raising the level of abstraction from the transistor level to the register-transfer level [Lie84]. Having been used for almost half a century, these HDLs are well-supported by the stable standards, industry-grade commercial HDL compilers, as well as decades of engineering training and practice. Figure 1.1(a) shows the HDL workflow where the designer: manually writes both the RTL design under test (DUT) and test bench (TB) in Verilog; compiles the DUT and TB into a simulator; uses the simulator to iteratively verify and evaluate the DUT; and eventually pushes the DUT through an FPGA/ASIC toolflow. The *iterative development cycle* (i.e., designer $\rightarrow$ DUT $\rightarrow$ simulation $\rightarrow$ designer) is contained within a single language.

However, the limited general-purpose programming capabilities and parametrization power provided by HDLs makes it difficult to effectively create highly parametrized and configurable hardware generators. Even though the HDL standards are constantly receiving upgrades that make these HDLs slightly more object-oriented (e.g., SystemVerilog [SDF06] superceded Verilog IEEE standard in 2008 [iee21]), those are mostly incremental changes that do not change the static nature of the language. For testing and verification, HDLs only provides limited high-level programming

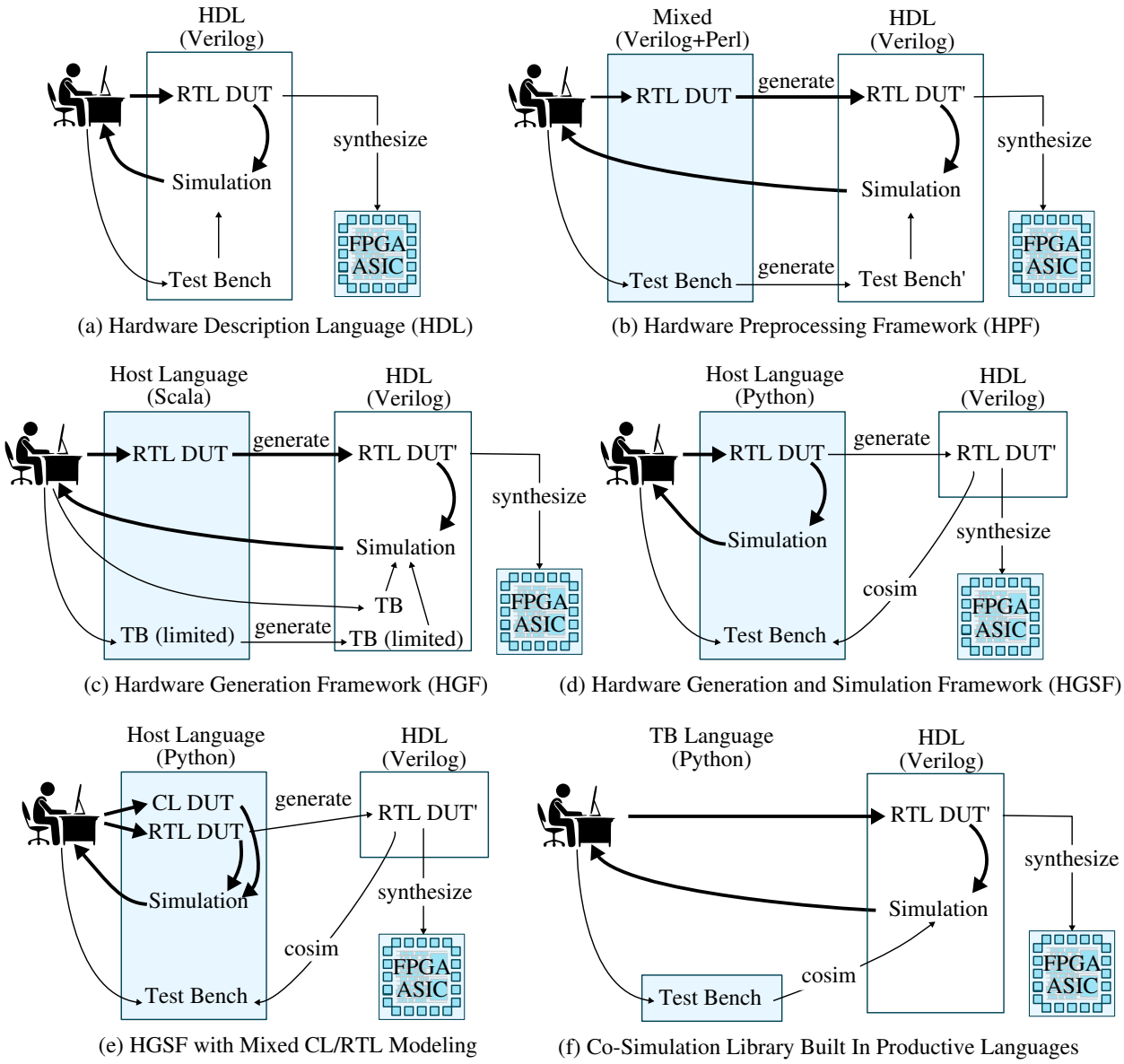


Figure 1.1: Workflows of Different Generations of Productive Hardware Modeling Methodologies – RTL = register-transfer level; CL = cycle-level; DUT = design under test; DUT' = generated DUT; TB = test bench; TB* = TB with limited functionality; TB' = generated TB; Sim = simulation.

capabilities for effectively building test benches. Although HDLs sometimes resort to external C++ libraries (e.g., VPI [DPR96] in Verilog) to incorporate more high-level programming capabilities, they are still far from sufficient to accommodate the rapidly evolving algorithms in modern specialized accelerators.

Hardware Preprocessing Frameworks (HPF) – Early attempts to make HDLs more productive focused on building hardware preprocessing frameworks that intermingle a high-level lan-

guage for macro-processing and a low-level HDL for logic modeling (e.g., Scheme mixed with Verilog in Verischemelog [JB99], Perl mixed with Verilog in Genesis2 [SAW$^+$10]). Figure 1.1(b) shows an HPF workflow using Genesis2 [SAW$^+$10] where the designer: writes the DUT and TB in a mix of Perl and Verilog; uses Perl to preprocess the DUT and TB into pure Verilog; and then transitions to the traditional HDL workflow. The use of a high-level language provides parametrization power and high-level constructs that HDLs lack. The simulation is still done in Verilog, which means the credibility of industry-standard HDLs is preserved.

The major drawback of mixed-language HPFs is that the high-level language only acts as a simple text preprocessor without any understanding of hardware semantics. This creates an abrupt semantic gap in the hardware description, since engineers must simultaneously design, verify, and reason about designs written in a high-level language (for parameterization, static elaboration, test bench generation) and a low-level HDL (for behavioral modeling). As shown in Figure 1.1(b), the *iterative development cycle* (i.e., designer $\rightarrow$ DUT $\rightarrow$ generated DUT $\rightarrow$ simulation $\rightarrow$ designer) stretches across two languages. For testing and verification, the designers cannot use high-level data structures provided by the high-level language at runtime, because these frameworks only use the high-level languages for macro processing. Thus the designers have to use the same testing/verification flow as HDLs.

Hardware Generation Frameworks (HGF) – Taking one step forward, true hardware generation frameworks address the semantic gap found in HPFs by completely embedding parameterization, static elaboration, test bench generation, *and* behavioral modeling in a unified high-level "host" language (e.g., Haskell in Lava [BCSS98], standard ML in HML [LL00], Scala in Chisel [BVR$^+$12], Python in Stratus [BDM$^+$07], PHDL [Mas07]). Figure 1.1(c) shows an HGF workflow using Chisel [BVR$^+$12] where the designer: writes the DUT and TB in Scala using the Chisel library; executes the Scala program to generate a Verilog DUT and TB; and then transitions to the traditional HDL workflow. Being able to describe hardware using a single embedded domain-specific language (EDSL) means the high-level language features can be fully utilized during the hardware generation process, which eliminates the mixed-language description in HPFs.

However, HGFs still generate and simulate low-level HDL code. This creates a modeling/simulation language gap that may require the designer to frequently cross language boundaries during iterative development. A few HGFs are able to generate test benches but usually with limited functionalities, since not all high-level code is translatable to HDL. For example, it is difficult to

translate the manipulation of Python deque/dictionary data structures to Verilog. Designers need to manually write more sophisticated Verilog TBs to run complex tests. In summary, HGF workflows still create a potentially frustrating language gap by stretching the iterative development cycle across multiple languages (i.e., designer → DUT → generated DUT → simulation → designer, as shown in Figure 1.1(c)).

Hardware Generation and Simulation Frameworks (HGSF) – The drawbacks in HPFs and HGFs have inspired researchers to build completely unified hardware generation and simulation frameworks (HGSFs) where parameterization, static elaboration, test bench generation, behavioral modeling, *and* a simulation engine are all embedded in a single general-purpose high-level language (e.g., Java in JHDL [BH98], Haskell in CλaSH [BKK+10], Python in MyHDL [Dec04], PyRTL [CTD+17], Migen [mig], PyHDL [HMLT03]). Figure 1.1(d) shows an HGSF workflow using PyMTL [LZB14] where the designer: writes the DUT and TB completely in Python using the PyMTL library; uses Python-based simulation to verify and evaluate the DUT; iteratively improves the design within Python; occasionally co-simulates the generated HDL code with the Python test bench; and only transitions to the traditional HDL workflow to push the DUT through an FPGA/ASIC toolflow. A key feature of HGSFs is the ability to use a simulation engine written in the host language to drastically reduce the iterative development cycle and eliminate any semantic gap. The designer avoids crossing any language boundaries for development, testing, and evaluation, and can use the complete expressive power of the host language for verification, debugging, instrumentation, and profiling. Python has been chosen by most modern HGSFs as the host language because Python is currently the most popular programming language for its high productivity and its large open-source community [pyp21].

By rapidly iterating inside the high-level language, HGSFs are able to realize the agile hardware manifesto [LWC+16]. Moreover, it is worth noting that simulating inside a high-level language brings up a synergy between RTL modeling methodologies and cycle-level modeling methodologies. Computer architects often leverage hardware emulators/simulators to build cycle-level (CL) models of the hypothetical hardware architecture [You07, BBB+11, PACG11, RCBJ11, BYF+09, SBM+19, AKPJ09, LSC+10, boo11]. Compared to RTL models, CL models include less hardware detail, only capture the approximate timing behavior and number of critical hardware events, and usually cannot be converted to hardware. However, the biggest advantages of CL models are the faster simulation speed and easier modification/enhancement. This allows computer architects to

explore and evaluate novel architectural/microarchitectural techniques using classic software engineering paradigms including object-oriented programming, high-level programming languages, and high-level data structures. For example, a CL cache model can be a Python class that models the tag arrays using double-ended queues, which makes it easy to explore the cache replacement policy. By enabling CL modeling and CL/RTL composition, the iteration inside the high-level language can be faster, and gradually replacing CL blocks with newly developed RTL blocks makes it easier to: (1) maintain the integration tests, end-to-end tests, and performance regressions, and (2) steadily improve the model fidelity of the whole design. Figure 1.1(e) shows an enhanced version of the HGSF flow where CL models and RTL models can be co-simulated and iteratively improved in the host language. SystemC [Pan01] and PyMTL [LZB14] are two frameworks that supports mixed CL/RTL modeling and composition in a single language.

Moreover, simulation in Python-based HGSFs appears to be very useful for testing and verification for specialized accelerators. Python-based programming makes it relatively easy to implement the algorithms to create golden reference models. For example, commonly used machine-learning libraries (e.g., Tensorflow [ABC+16], PyTorch [PGM+19], TVM [CMJ+18]) are built in Python, which can be leveraged for testing machine-learning accelerators.

These opportunities make Python-based HGSFs very compelling for reducing the NRE costs in the era of heterogeneous SoCs.

Co-Simulation Libraries Built in Productive Programming Languages – Embedding the modeling/simulation of hardware inside productive languages is not the only way to leverage productive languages for hardware design. As previously mentioned, Verilog Procedural Interface (VPI) enables a *Verilog simulator* to co-simulate models built in productive high-level languages with Verilog models, as long as these languages can be integrated with C/C++. Engineers have been building co-simulation libraries to improve the productivity of building *test benches* instead of designs, as those models built in high-level languages usually do not include RTL semantics. Figure 1.1(h) shows the workflow of using a co-simulation library with Verilog models. CocoTB is a representative co-simulation framework that builds hook functions in Python and triggers them in the Verilog simulator events using Python/C++ integration mechanisms. Such co-simulation libraries only target test benches and golden reference models, while complicating the ability to leverage the full power of the high-level language. Also, if the model is built using an HPF/HGF,

the workflow requires the designer to deal with at least three different languages (e.g., Verilog + Perl + Scala) at the same time, which can be cumbersome.

1.2 Key Challenges in HGSFs

This thesis aims to address the following four challenges in the state-of-the-art hardware generation and simulation frameworks.

Improving the Flexiblity and Extensibility of HGSFs – Many of the aforementioned state-of-the-art productive hardware modeling frameworks are relatively monolithic. The lack of flexiblity and extensibility in these monolithic frameworks makes it much more difficult to perform continuous development for feature extensions after the initial release. This is because those frameworks leverage various meta-programming mechanisms to create a handy and convenient embedded domain-specific language, but fail to separate the implementation of these mechanisms. There have been attempts to design intermediate representations (IR) [IKL+17, MMB+18] for hardware constructs. However, these hardware IRs are mostly describing the hardware netlists *after* high-productivity modeling. In other words, the framework does not benefit from the existence of these IRs, and the extensibility of the modeling framework is limited by what is processed before turning the description into the IR representation. The fact that every designer has their own evolving wishlist of features imposes great challenges on the HGSF framework designer to create flexible and extensible hardware modeling frameworks.

Unifying CL and RTL Modeling to Achieve High Model Fidelity With Little Effort – There is a modeling/simulation mechanism gap between RTL and CL modeling in state-of-the-art RTL and CL modeling methodologies. RTL modeling has well-established discrete-event simulation semantic. For example, Verilog RTL simulators leverage sensitivities of logic blocks and direct assignments to establish a graph containing intra-cycle operations on the signals. These simulators either use an event queue to dynamically trigger intra-cycle logic based on sensitivity, or statically schedule and then execute the logic in topological sort order. In contrast, a CL simulator's modeling mechanism can be arbitrary, because by definition CL models just need to "approximately" capture the timing of the RTL model. As a result, there is not a single widely adopted CL modeling mechanism. In state-of-the-art CL simulators, the model fidelity is usually improved by manually

scheduling CL processes, and then looking at traces to perform result-driven reverse engineering. This mechanism gap is more prominent when an HGSF wants to incorporate CL modeling to take advantage of the high-level language productivity. In order to fully utilize CL modeling in an HGSF to reduce NRE costs, a unified abstraction of RTL and CL modeling is a preferred solution. This requires standardizing CL modeling by representing and scheduling CL and RTL processes in a compatible way. However, state-of-the-art HGSFs [LZB14, Pan01] only support coarse-grained CL/RTL composition by combining the CL and RTL portions in an ad-hoc way. They still use different modeling mechanisms for CL and RTL parts, and the composition of the CL/RTL boundary is forced to have inter-cycle effects instead of allowing intra-cycle behavior, which impairs the model fidelity. Unifying CL and RTL modeling remains a challenge for HGSF designers to address.

Closing the Simulation Performance Gap in HGSFs – Different from HDLs/HPFs/HGFs which perform simulation using HDL simulators, Python-based HGSFs include a simulation engine *in pure Python*. However, most Python-based HGSFs have dismal performance with CPython (the de-facto Python interpreter) compared to HDL simulators. This is because the dynamic typing system in Python requires the Python program to be dynamically interpreted instead of statically compiled. This simulation performance gap partially undermines the productivity benefits obtained from using Python-based HGSFs. Previous work attempts to leverage PyPy, the only available Python interpreter with tracing just-in-time (JIT) optimization. The speedup of simply using PyPy over CPython failed to close the gap. Previous work also performs Python-C++ co-simulation where the hardware design logic is translated into low-level code and statically compiled into a C++ library. This accelerates the hardware simulation, but the simulation performance bottleneck is still in the Python portion of the execution. Moreover, co-simulating Python and C++ brings back the semantic gap, as the signals values in the C++ portion are not directly observable in the Python portion without introducing significant overheads. It is also impossible to insert any non-translatable code in the logic blocks written in Python, which undermines the productivity promises. Closing the simulation performance gap in native Python execution remains challenge for HGSFs to address.

Reducing Testing/Verification Time for Agile Hardware Design Flows – The standard hardware testing/verification methodology is constraint-based random testing on input values using the Universal Verification Methodology (UVM) and SystemVerilog, which unfortunately does not find

many use cases outside industrial chip-design teams. Academic research groups and open-source hardware teams usually cannot afford to have dedicated verification teams, where the verification engineers have many years of experience in these commercialized UVM methodologies. They have to follow an agile test-driven design approach stemming from the open-source software community, where the designer is also responsible for creating and distributing the corresponding tests. HGSFs built in productive languages provide a good starting point to productively develop, iterate, distribute, and collaborate on hardware design blocks. However, besides the obvious benefits of being able to quickly create sophiscated test benches and golden models, leveraging the unique open-source communities of the HGSF host languages to reduce hardware testing and verification time is still a challenge awaiting joint efforts from both the HGSF developers and the HGSF users to address.

1.3 Thesis Overview

This thesis addresses the aforementioned hardware modeling challenges in Section 1.2 using an engineering research approach. After formulating each challenge into a well-defined research problem, I propose solutions to each research problem in Chapter 2–5. Figure 1.2 is an illustration of the thesis work where each solution addresses a challenge within the HGSF workflow, along with the corresponding first-author publications. Moreover, to demonstrate that these proposed novel techniques are also realistic, practical, and useful for engineering practices, I built PyMTL3, a novel hardware generation and simulation framework which implements all the novel techniques. The framework has been used to facilitate other research projects, engineering projects, chip tape-outs, and course lab assignments.

Chapter 2 introduces native in-memory intermediate representation (NIMIR) as a systematic approach to address the challenge of building extremely flexible and extensible hardware modeling frameworks, and discusses PyMTL3, a realistic HGSF I built using the NIMIR approach. NIMIR is a novel approach to build hardware generation and simulation frameworks (HGSF) that can be modularly maintained by different developers, easily enhanced by a growing designer community, and flexibly serve as a research platform. NIMIR separates the framework into three parts: domain-specific language implementation (front-end), in-memory data structure exposed through APIs (IR), and passes that invoke the APIs to analyze, instrument, and transform the in-memory model

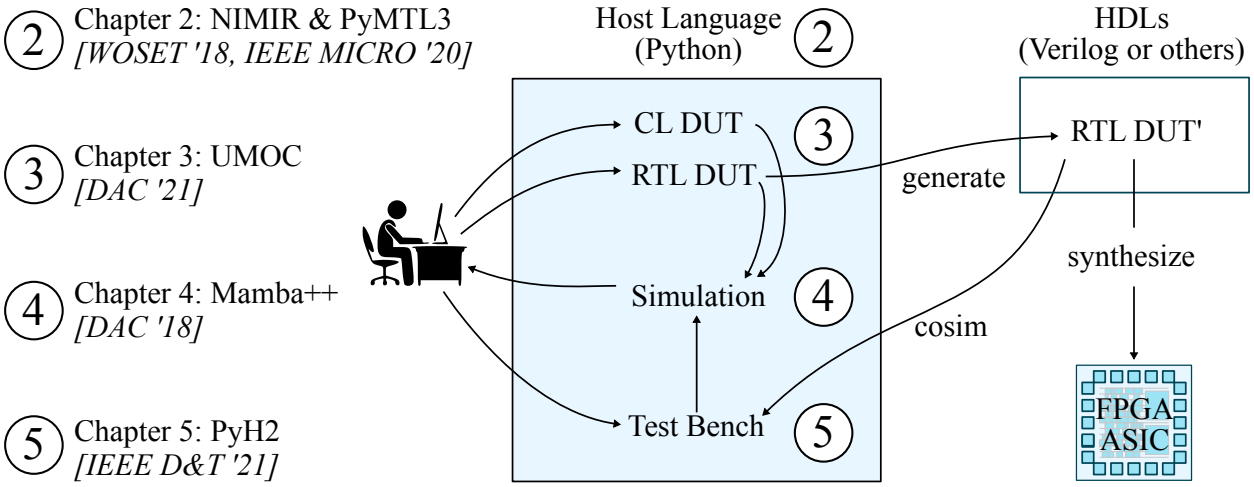


Figure 1.2: Thesis Overview and Breakdown in the HGSF Workflow – Section 1.2 discusses four challenges in hardware modeling methodologies. These challenge vividly corresponds to different parts of the HGSF workflow: the framework itself, CL/RTL modeling abstraction, simulation, and testing. Each chapter of the thesis corresponds to my work that solves each challenge. I also attach my first-author publications corresponding to each chapter.

(back-end). PyMTL3 is the first framework built under this NIMIR approach and demonstrated extensibility in many use cases. I illustrate the details of the PyMTL3 framework in this chapter along with the NIMIR concept. I also present a case study on adding new modeling primitives, new data structure and APIs, and backend passes that enable simulating those primitives without affecting existing framework functionalities in PyMTL3. This work was published in an IEEE Micro Special Issue on Agile and Open-Source Hardware (2020) [JPOB20], and I was the lead author of this work. An early version of this work was published in First Workshop on Open-Source EDA Technology (WOSET 2018) [JTB18]. In practice, my colleagues and I have been adding various passes to the PyMTL3 framework and built an ecosystem of various open-source hardware IPs. PyMTL3 has been used in Cornell University's ECE 5745 course to replace the previous PyMTL2 framework. PyMTL3 has also been used in various GF 14nm chip tapeouts.

The rest of the thesis chapters propose generic mechanisms. As PyMTL3 is designed to be extremely flexible and extensible, the PyMTL3 framework actually manages to implement all the proposed mechanisms and enables the designer to leverage those techniques for hardware modeling. I will be using framework and design code implemented using PyMTL3 as a concrete running example to provide readers with embodiment of the generic mechanisms.

Chapter 3 presents unified modular ordering constraints (UMOC), a novel technique to unify signal-based RTL modeling and method-based CL modeling in HGSFs. UMOC provides a uni-

fied view for general-purpose CL and RTL modeling and enables automatically scheduling all the CL/RTL hardware processes with designer-specified (CL) or inferred (RTL) *local* constraints without manually specified global intra-cycle ordering of hardware processes. The designer can reason about intra-cycle execution order in a systematic and modular way for CL processes and RTL processes, encapsulate CL ordering constraints in components, and reuse them for other design blocks. UMOC is able to achieve high model fidelity for CL models and allows CL models to be seamlessly composed with RTL models. This work will be published at the 58th Design Automation Conference (DAC 2021) [JOPB21], and I am the lead author of this work. UMOC primitives and scheduling passes have been implemented in PyMTL3 as a key component of the PyMTL3 framework. CL/RTL mixed-level modeling using UMOC has also successfully deployed in the ASIC design course at Cornell University.

Chapter 4 addresses the simulation performance gap in native Python using a combination of JIT-aware HGSF design techniques, and HGSF-aware JIT optimization techniques. JIT-aware HGSF design techniques include how we design the simulation mechanisms in the HGSF to have code structures that the JIT engine can more effectively optimize. HGSF-aware JIT optimization techniques involve customizations and optimizations of the underlying JIT engine based on properties of hardware simulations. As we believe that Python-based HGSFs are important, we mostly focus on Python3 and PyPy (the only JIT compiler for Python). Moreover, this work sheds light on the simulation performance optimization of any Python-based HGSF (not limited to PyMTL3). The static scheduling part of the work was published at the 55th Design Automation Conference (DAC 2018) [JIB18], and I was the lead author of this work. The hierarchical scheduling part of the work is currently unpublished. Hierarchical scheduling takes the insights obtained from the DAC work and uses a more comprehensive algorithm to support practical situations such as graphs that cannot be statically scheduled and Verilog co-simulation which inevitably introduces cyclic dependencies. The simulation mechanisms of the PyMTL3 framework deploys the JIT-aware HGSFs techniques, and we also customize PyPy, the state-of-the-art tracing JIT compiler for Python, to deploy the HGSF-aware JIT techniques. Aside from research projects, PyMTL3 and the hierarchical scheduling passes have also been successfully deployed in the ASIC design course at Cornell University to significantly boost the simulation performance.

Chapter 5 lays out our vision for verifying open-source hardware IPs in the context of hardware generation and simulations frameworks. Leveraging Python, hypothesis, and PyMTL3, I present

three techniques to test hardware generators (PyH2G), processors (PyH2P) and object-oriented hardware data structures (PyH2O). Testing the hardware generator involves randomizing both the test case and the parameter, and co-shrinking them together to find the smallest failing design instance in the parameter space and the shortest failing test case. Testing processors requires randomizing the control flow patterns and arithmetic instructions. Testing object-oriented hardware data structures combines a novel scheduling mechanism (implemented as another scheduling pass in the PyMTL3 framework) based on UMOC to advance simulation "steps" upon method calls, and hypothesis stateful testing to produce a minimal sequence of transactions. This work was published in an IEEE Design & Test Special Issue on Open-Source EDA (2021) [JOP+20], and I was the co-first author of this work as the visionary of future verification directions, and the PyH2O contributor. PyH2 also showcases the synergy of open-source hardware and open-source software, and sheds light on the future verification methdologies enabled by HGSFs built in a productive language with a large open-source software community.

This thesis makes the following technical contributions:

- I propose native in-memory intermediate representation (NIMIR), a novel approach to build flexible and extensible hardware modeling frameworks. To demonstrate the practicality of NIMIR, I built PyMTL3, a new hardware modeling framework, from the ground up using NIMIR.
- I propose unified modular ordering constraints (UMOC), a novel technique to unify signal-based RTL modeling and method-based CL modeling. To demonstrate the practicality of UMOC, I implemented UMOC primitives and scheduling passes in PyMTL3, and built various hardware IPs in PyMTL3 using UMOC primitives.
- I propose Mamba++, a set of techniques to close the simulation performance gap in hardware generation and simulation frameworks. To demonstrate the practicality of Mamba++, I have implemented Mamba++ techniques in PyMTL3 as passes. Mamba++ passes and the modified PyPy JIT compiler have been deployed in production.
- I present PyH2, our vision for a novel hardware testing methodology that leverages open-source software. PyH2 includes three different testing approaches for highly parametrized hardware design generators, processors, and hardware data structures.

This thesis is also a contribution to the ongoing open-source hardware and open-source electronic design automation (EDA) movements. Other open-source HGSFs can take inspiration from the proposed techniques which are not specific to PyMTL3. However, from our experience in developing open-source hardware IPs, PyMTL3 is an ideal framework to jump-start the open-source hardware ecosystem.

1.4 Collaboration and Funding

I am very fortunate to have led several research projects throughout my Ph.D. career. I am really glad that I have the chance to collaborate with my brilliant colleagues from the Batten Research Group at Cornell University. Most importantly, my Ph.D. advisor Christopher Batten has been a major influencer throughout these years. I have had countless brainstorming sessions with him, which really supercharged these research projects.

The work on native in-memory intermediate representation (NIMIR) as a novel way to build hardware modeling frameworks is fueled by Peitian Pan. Peitian spent many hours building, refactoring, and even overhauling the RTLIR and translation passes in order to build a clean and elegant translation framework, which really demonstrated the power of the NIMIR architecture. Peitian also standardized the internal metadata data structure in the PyMTL3 NIMIR implementation.

The PyMTL3 framework has received contributions from many colleagues. Peitian Pan was the first developer (other than myself) to write PyMTL3 passes, and he even went above and beyond to create a translation pass framework and his own RTLIR. Yanghui Ou was the major contributor and helper for enriching the PyMTL3 standard library, as well as the first developer (other than myself) to deal with ordering constraints at the boundary between cycle-level and RTL components. Many of my colleagues from Batten Research Group took the initiative in building/distributing various PyMTL3 hardware IP blocks using PyMTL3, and even attempted to use PyMTL3 to facilitate chip tapeouts. Those first-hand development experiences turned into bug reports and feature requests to help improve the PyMTL3 framework. Dr. Cheng Tan and Yanghui Ou created the first PyMTL3 hardware IP `pymtl3-net` (PyOCN) which provides a realistic hardware generator use case for the PyMTL3 framework to improve upon. Moyang Wang, Eric Tang, and Xiaoyu Yan created `pymtl3-mem`, the blocking cache generator with software-centric cache coherence. Tuan Ta built `pymtl3-proc`, the modular RV32IMAF processor, extensively leveraging method-

base interfaces and modular directed testing. The RTL code of the BRG-portion of CIFER and Hammerblade tapeouts are all developed and tested using PyMTL3, and we even addressed the Verilog test harness problem by merely creating another 200-line pass. Christopher Torng, Khalid Al-Hawaj, Lin Cheng, and Dr. Shady Agwa joined to help organize the first PyMTL3 tutorial at the 46th International Symposium on Computer Architecture in Arizona, which turned out to be a big success.

The work on unified modular ordering constraints (UMOC) to unify CL and RTL modeling is fueled by Yanghui Ou. Yanghui implemented many CL/RTL boundary adapters for different interfaces and experimented with complicated scenarios with invalid loops going across the CL and RTL portions. Yanghui's work deepened our understanding in the equivalence of some CL and RTL semantics using method-based interfaces.

The original work on Mamba to close the simulation performance gap in Python-based hardware modeling frameworks would not have been possible without Berkin Ilbeyi's expertise in PyPy/RPython at the initial stage. Berkin provided insights into how the tracing-JIT engine and PyPy works, and solved the huge-page issues. Berkin also proposed trace breaking techniques to create loop structures suitable for JIT optimization. During the process of getting fast simulation performance in production, Mamba++ received useful guidance from Carl Friedrich Bolz and Lin Cheng on further improving the RPython Bits implementation and resolving Python3 specific issues.

The work on PyH2, our vision for open-source hardware verification, was co-led by Yanghui Ou and me, with help from Zac Hatfield-Dodds on `hypothesis`, Peitian Pan and Kaishuo Cheng on PyH2P, Dr. Cheng Tan on PyH2G, and Yixiao Zhang on PyH2O. Even though I wrote most of the submission to IEEE Design & Test, it was Yanghui's hard work on leveraging hypothesis to test hardware generators that shed light on all kinds of possibility of leveraging a random testing framework built for software to test hardware. Peitian and Kaishuo led the work on generating random instruction patterns and sequences to automatically test a PyMTL3 processor. Yixiao dedicated her MEng project to experimenting with hardware data structures and stateful hypothesis testing.

In terms of funding, this thesis was supported in part by Cornell Graduate School Fellowship, Richard E. Lunquist Graduate Award, NSF SHF Award #1527065, NSF CRI Award #1512937, AFOSR YIP Award #FA9550-15-1-0194, DARPA SDH Award #FA8650-18-2-7863, DARPA POSH

Award #FA8650-18-2-7852, DARPA CRAFT Award #HR0011-16-C-0037, a research gift from Xilinx, Inc., and the the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA. This work was also supported by equipment, tool, and/or physical IP donations from Intel, Xilinx, Synopsys, Cadence, and ARM. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation theron. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

CHAPTER 2
PYMTL3: A PRODUCTIVE AND EXTENSIBLE FRAMEWORK FOR HARDWARE MODELING, SIMULATION, AND VERIFICATION

The first key challenge in state-of-the-art hardware modeling frameworks, as mentioned in Chapter 1, is the lack of flexibility and extensibility to accommodate the ever-growing feature wishlist. In this chapter, I propose native in-memory intermediate representation (NIMIR), a novel and systematic approach to build productive hardware modeling frameworks. NIMIR enables the framework to accommodate new ideas from different angles of computer architecture, electronic design automation, and even circuit design in a collaborative community. Then, I present the PyMTL3 framework, the first framework built under NIMIR. PyMTL3 is a productive and extensible framework for hardware modeling, simulation, generation, and verification.

2.1 Introduction

Due to the breakdown of transistor scaling [DGY+74] and the slowdown of Moore's Law [Moo65], there has been an increasing trend towards energy-efficient system-on-chip (SoC) design using heterogeneous architectures with a mix of general-purpose and specialized computing engines. Heterogeneous SoCs [WJM08] emphasize both flexible parameterization of a single design block and versatile composition of numerous different design blocks, which have imposed significant challenges to state-of-the-art hardware modeling and verification methodologies.

To respond to these challenges, computer engineers are augmenting or even replacing traditional domain-specific hardware description languages (HDLs) with productive hardware development frameworks empowered by high-level general-purpose programming languages such as C++, Scala, Perl, and Python. *Hardware preprocessing frameworks* intermingle a high-level language for macro-processing and a low-level HDL for logic modeling (e.g., Scheme mixed with Verilog in Verischemelog [JB99], Perl mixed with Verilog in Genesis2 [SAW+10]), which enables more powerful parametrization, yet creates an abrupt semantic gap in the hardware description. *Hardware generation frameworks* completely embed parametrization and logic description in a unified high-level "host" language (e.g., Haskell in Lava [BCSS98], standard ML in HML [LL00], Scala in Chisel [BVR+12], Python in Stratus [BDM+07], PHDL [Mas07]), but still generates and sim-

ulates low-level HDL code. This requires test benches to be written in the low-level HDL, which creates a modeling/simulation language gap that may require the designer to frequently cross language boundaries during iterative development. All these challenges have inspired completely unified *hardware generation and simulation frameworks* where parametrization, static elaboration, test benches, behavioral modeling, *and* a simulation engine are all embedded in a general-purpose high-level language (e.g., Java in JHDL [BH98], Haskell in CλaSH [BKK$^+$10], Python in MyHDL [Dec04], PyRTL [CTD$^+$17], Migen [mig], PyHDL [HMLT03]). *High-level synthesis* (HLS) is an alternative approach that seeks to automatically synthesize software-oriented programs written in C++ into low-level HDL implementations [CCA$^+$11, CLN$^+$11]. We see HLS as complementary to the emerging trend towards hardware generation and simulation frameworks, since any realistic SoC will require a mix of blocks well-suited to HLS (e.g., well-structured data-processing blocks, low-performance control blocks) and blocks that require designers to control more hardware details (e.g., processors, memory hierarchies, networks-on-chip, complex accelerators).

At the same time, computer architects are using cycle-level (CL) modeling methodologies such as SystemC and Cascade [GTBS13] to facilitate rapid design-space exploration of large SoCs before creating RTL implementations. When moving from CL to RTL, the ability to support seamless *multi-level modeling* (i.e., mix and match RTL models with CL models) provides significant productivity benefits. For each individual design block, the CL model can serve as the golden reference model, which means all the unit tests can be reused to test the RTL model. Moreover, in a development flow with continuous integration, gradually replacing existing CL blocks with newly developed RTL blocks in a large design while maintaining the integration tests, end-to-end tests, and performance regressions significantly reduces the integration effort and steadily improves the performance accuracy of the overall model.

To further improve the productivity of both hardware designers and computer architects, we have built PyMTL3, an open-source Python-based *hardware modeling, generation, simulation, and verification framework*. PyMTL3 is a brand new hardware modeling framework instead of a regular update to its predecessor PyMTL2 [LZB14]. The design philosophy of PyMTL3 incorporates two important takeaways from PyMTL2: (1) *modularity* of the framework is the key to creating a vibrant and evolving hardware development ecosystem; and (2) *interoperability* with other open-source tools is the key to achieving widespread adoption. Motivated by these two key takeaways, I propose native in-memory intermediate representation (NIMIR), a novel approach to

build extensible hardware modeling frameworks. NIMIR separates a hardware modeling framework into three parts: front-end domain-specific language, the native in-memory intermediate representation, and back-end passes. Section 2.2 describes the NIMIR architecture design in depth. Implemented from the ground up, PyMTL3 is the first framework that adopts the NIMIR architecture and demonstrates strong extensibility. In terms of modeling features, PyMTL3 maintains the key features of PyMTL2, and also includes a series of novel features: unified modular ordering constraints (UMOC) for seamless multi-level modeling across register-transfer level (RTL), cycle level (CL) and functional level (FL); a new parameter configuration system; first-class method-based interfaces; polymorphic interface connections; and faster simulation performance using the Mamba++ techniques under `PyPy` just-in-time compiler. PyMTL3 leverages the latest Python 3 features where PyMTL2 only works on Python 2. Section 2.3 presents the PyMTL3 framework in-depth, discussing PyMTL3 embedded DSL, PyMTL3 NIMIR, and PyMTL3 passes. Section 2.4 includes a developer's case study on supporting delay-annotated gate-level modeling in PyMTL3. The framework developer adds eDSL modeling primitives, NIMIR data structures/APIs, and scheduling passes to support the new modeling feature without affecting any existing features. This demonstrates that PyMTL3 enables the researchers to quickly explore a variety of new ideas in hardware modeling methodology research with no impact to the rest of the PyMTL3 framework.

PyMTL3 has been extensively used in graduate courses at Cornell University, and two large-scale chip tape-outs in GF 14nm. Many PyMTL3 IPs have been built as part of the PyMTL3 ecosystem. Moreover, the recent open-source hardware movement implies that developing, open-sourcing, and collaborating on hardware generators is a compelling solution to increase the reuse of highly parametrized and thoroughly tested hardware blocks across academia and industry. community. However, the general lack of high-quality open-source hardware designs and hardware verification methodologies have been a major concern that limits the widespread adoption of open-source hardware. Section 2.5 discusses PyMTL3's potential to jump start the open-source hardware ecosystem.

2.2 Native In-Memory Intermediate Representation

In this section, I propose native in-memory intermediate representation (NIMIR), a novel approach to systematically build flexible and extensible hardware modeling frameworks. The NIMIR framework architecture forms the foundation of the PyMTL3 framework.

2.2.1 Motivation

Most hardware designers have their evolving wishlist of new features that can improve their productivity. The ideal hardware modeling framework should allow the designers to not only select "flow steps" to form their own suitable workflow, but also accommodate the ever-growing feature wishlist with lightweight changes to the existing codebase. However, existing hardware modeling frameworks are not flexible and extensible enough to fulfill such purposes. The fundamental reason is that almost all aforementioned hardware modeling frameworks (see Section 1.1) are built in a monolithic way. Those frameworks leverage various meta-programming mechanisms to create a convenient embedded domain-specific language, but fail to separate the implementation of these mechanisms. For example, PyMTL uses the Python metaclass to implement the Verilog black-box import feature. The Python metaclass mechanism is notoriously difficult to reason about and will lead to unpleasant error messages if not used correctly. It becomes much more difficult to perform continuous development for feature extensions when the framework requires developers to fully understand the intricacies of these mechanisms (even if they are implementing unrelated features). Also, PyMTL's Verilog import can only happen at a specific time during elaboration between specific steps. Otherwise the whole elaboration process will break down. These kinds of assumptions significantly limit the flexiblity of the framework in terms of adding new features without breaking existing workflows.

As suggested by modern software engineering practices, modularity is the key to improve the flexibility and extensibility of a framework. There have been attempts to design hardware intermediate representations (IR) [IKL+17, MMB+18] to separate the hardware description from processing the elaborated model. However, these hardware IRs are mostly describing the hardware netlists *after* the high-productivity modeling phase. Although the netlist analysis and optimization process significantly benefit from having such hardware IRs, the hardware modeling framework

itself does not benefit from the existence of these IRs. In fact, the extensibility of the modeling framework is limited by what is processed before turning the description into the IR representation.

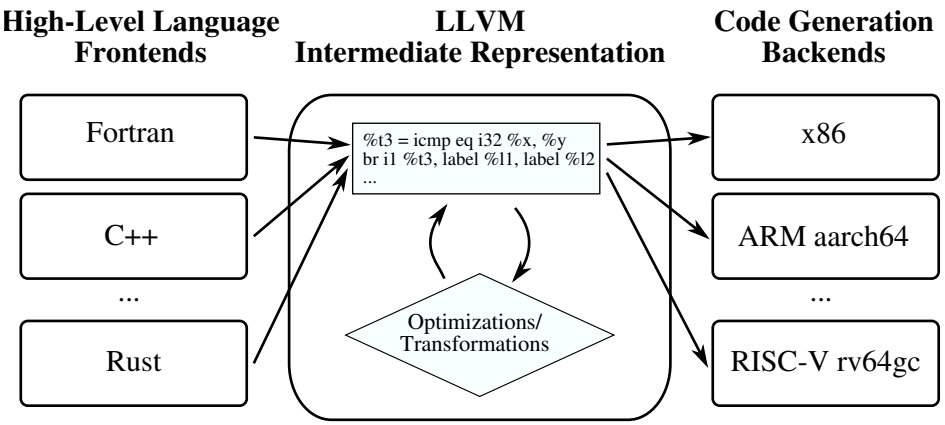
We conclude that the community is in need of a novel approach to modularize hardware modeling frameworks.

2.2.2 NIMIR Architecture

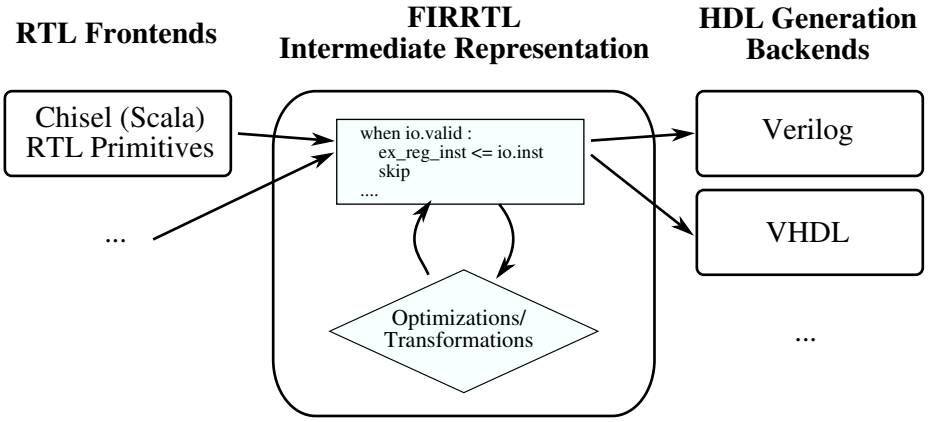
The proposed NIMIR architecture is inspired by LLVM, a successful modular compiler infrastructure project in the open-source software community [LA04]. As shown in Figure 2.1(a), the LLVM architecture's front-ends compile code of different programming languages (e.g., Fortran, C++, Rust) into the same LLVM intermediate representation (IR). The IR is stored as an in-memory data structure during execution but also has a serialized text form. Then, optimization passes can be applied on the IR representation to analyze/mutate the IR. Finally, LLVM supports multiple backends (e.g., x86, ARM, RISC-V) for code generation. Such modular architecture enables developers/researchers with different focuses to work in different parts of the LLVM framework without affecting the rest of the framework. As a result, LLVM has been continuously developed for about twenty years, receiving contributions from both industry and academia.

Inspired by the frontend/IR/backend division in LLVM, I design the NIMIR architecture that separates a hardware modeling framework into three parts: frontend embedded domain-specific language (eDSL), intermediate representation (IR), and backend passes. Previous hardware IRs such as FIRRTL has very similar architecture to LLVM as shown in Figure 2.1(b). Figure 2.1(c) illustrates the architecture of NIMIR. Note that the NIMIR architecture is similar to LLVM in spirit but different in details. NIMIR targets hardware modeling frameworks built in a specific language such as Python or Scala. The designer will not leave the language environment for development until the HDL code generation process is invoked. The words "native" and "in-memory" in NIMIR means NIMIR does not have serializable text forms and are only captured in the system memory as the native language data structures. This is because cycle-level models and functional-level models are essentially normal Python code; serializing the IR is simply serializing the Python code. In summary, NIMIR provides a *model-level* view of the whole design hierarchy for not only the RTL circuits, but also CL/FL hardware processes.

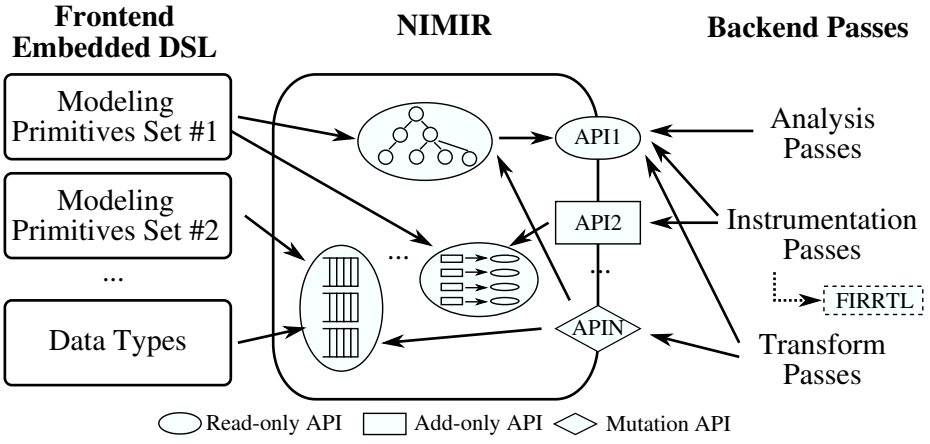
NIMIR Embedded Domain-Specific Language – The NIMIR embedded domain-specific language involves a series of modeling primitives and data types. Since the eDSL primitives are



(a) LLVM Architecture



(b) FIRRTL Architecture



(c) NIMIR Architecture

Figure 2.1: LLVM Architecture vs. FIRRTL Architecture vs. NIMIR Architecture – The LLVM architecture and FIRRTL architecture both have a text-based intermediaterepresentation

basically user interfaces to construct hardware, the framework developer will need to leverage the language's features to create designer-friendly primitives to maximize the productivity of hardware designers. A good example is that MyHDL and PyMTL leverage Python `@xxx` decorators to mark Python functions as hardware process instead of using verbose API calls. When the primitives are invoked, the underlying implementation will analyze and store the content in NIMIR. For example, the user invokes the hardware component definition primitives to create a hardware component. The underlying primitive implementation may collect the class and store it into the list of available hardware component classes.

Note that as multiple modeling primitives can be designed to store the same metadata, NIMIR opens opportunities for supporting different sets of modeling primitives (i.e., different DSLs) just like LLVM's various language frontends, without modifying the NIMIR or passes.

NIMIR Intermediate Representation – As previously mentioned, the NIMIR intermediate representation does not have a text form. Instead, it is a systematic organization/centralization of in-memory data structures constructed and elaborated from the hardware model. When the user invokes NIMIR DSL primitives, the implementation of these primitives should collect, organize, and store the specified hardware constructs such as ports, wires, combinational blocks, and sequential blocks in the hardware component. All these stored data structures are centralized in the NIMIR namespace of the hardware model, and the models expose public methods (i.e., APIs) that systematically manage these data structures. There are three types of APIs: read-only, add-only, and mutation. For example, the ports of a hardware component in the model hierarchy can be stored as a list in NIMIR and queried by the `get_ports` read-only API. Passes that add functionality to the model will call an add-only API to attach those newly created metadata to the model. Passes that systematically replace some modules with other modules will call mutation APIs.

Note that NIMIR is not a substitute for hardware IRs. It is totally suitable for NIMIR and hardware IRs to co-exist in the same development flow. The designer can implement passes that translate RTL code described in the DSL to low-level HDL/IR code. Then, the workflow of hardware IRs can take over and optimize the netlists. This resembles a two-level IR structure where NIMIR is the IR for the modeling framework and then lowered to the low-level hardware IR for netlist processing as shown in Figure 2.1(c).

NIMIR Passes – Because hardware modeling frameworks are far more (e.g., modeling, simulation, HDL generation, etc) than simply compiling/optimizing IR code, the concept of NIMIR

passes is more general than LLVM's optimization passes which only analyze and transform code. NIMIR passes are systematic programs that interact with the NIMIR intermediate representation. Specifically, a pass should call the three types of APIs provided by NIMIR to obtain metadata of the hardware design hierarchy, add useful functionality, or mutate the elaborated hardware model. Passes should be modular by themselves in the sense that the user can skip unneeded passes and only apply a subset of passes. Hence, the passes must be designed in a way such that adding new passes or modifying existing passes does not break the functionality of unrelated passes. Enforcing this guideline significantly facilitates collaboration in the community.

Inspired by LLVM's pass categorization (analysis and transform), I categorize NIMIR passes into three categories:

- *Analysis passes* call read-only NIMIR APIs to simply analyze the NIMIR hardware model and generate useful outputs without any modification to NIMIR. Designers can implement their own net list analysis tools as analysis passes.
- *Instrumentation passes* call read-only and add-only APIs to enhance the model with additional functionalities without any modification to the hardware hierarchy. Simulation and HDL generation tools are typical instrumentation passes that add simulating facilities or HDL source to the hardware model.
- *Transform passes* call read-only and mutation APIs to mutate the hardware hierarchy by adding/removing/replacing part of the model. Transform passes are very helpful if the designer wants to add some debugging support without modifying the original HDL code.

The PyMTL3 framework follows this the pass categorization. Section 2.3.3 includes more details on concrete PyMTL3 passes of each category.

2.3 The PyMTL3 Framework

Figure 2.2(a) illustrates an example PyMTL3 workflow. The designer starts from developing a functional-level (FL) design-under-test (DUT) and test bench (TB) completely in Python. Then the DUT is manually refined to a cycle-level (CL) and/or register-transfer-level (RTL) model. The designer simulates and evaluates the DUT/TB composition, and debugs the FL/CL/RTL DUT

leveraging various tracing output. The designer can also leverage the PyH2 property-based testing framework to find minimal failing test cases. Meanwhile, the designer uses the existing analysis tools or creates new ones to assist iterative refinement. The designer may temporarily transform the hardware model to replace modules or add new logic without modifying the original design. After iterating in the pure-Python environment, the designer invokes translation backends to generate SystemVerilog code and import it back to PyMTL3 for co-simulation with the same TB. Finally, the designer can push the translated SystemVerilog code through an FPGA/ASIC toolflow, and use a prototype proxy that PyMTL3 generates based on the original DUT to test the FPGA/ASIC prototype using the same TB. Designers who only write SystemVerilog code can still benefit from most of the productive workflow steps through PyMTL3's SystemVerilog import. Computer architects may iterate more in CL modeling and only implement RTL for critical parts.

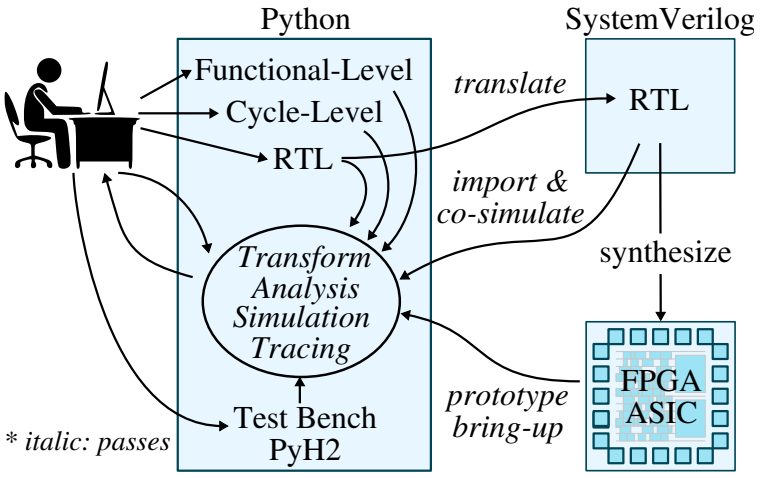
Figure 2.2(b) shows the software architecture of PyMTL3. The PyMTL3 embedded DSL exposes the modeling primitives to the designer for describing hardware, creating test benches, and configuring parameters. PyMTL3 is responsible for elaborating the hardware model and creating an native in-memory intermediate representation (NIMIR) that exposes APIs to query/modify the stored metadata of the whole hierarchical model. Then various PyMTL3 passes can analyze, instrument, and/or transform an elaborated PyMTL3 NIMIR model.

Lines 1–32 of Figure 2.3 show the PyMTL3 implementation of a registered incrementer unit and a parametrized N-stage registered incrementer using PyMTL3 embedded DSL primitives.

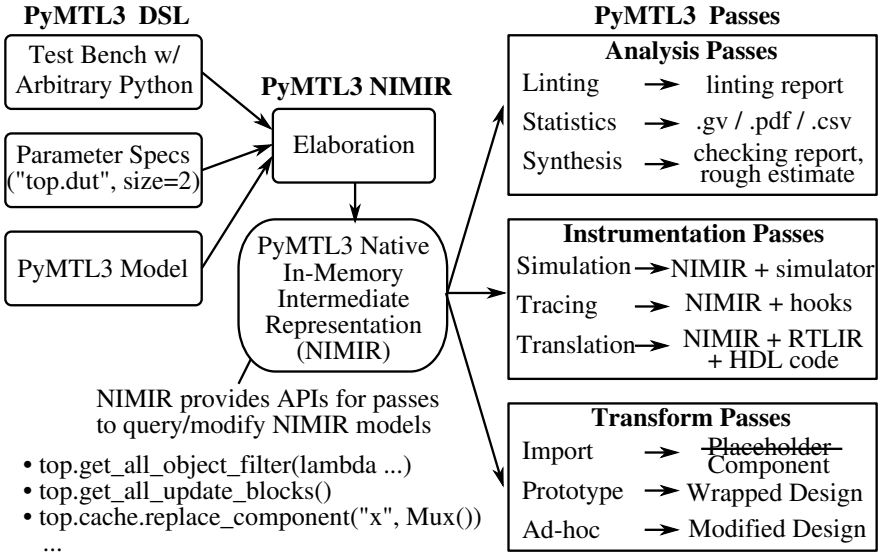
2.3.1 PyMTL3 Embedded DSL

PyMTL3's embedded DSL provides several distinctive modeling features that are not found in existing frameworks (including PyMTL2).

Unified Multi-Level Modeling and Scheduling – PyMTL3 provides three sets of primitives for FL, CL, and RTL modeling. FL/CL update blocks communicate through methods, and RTL update blocks communicate through signals. PyMTL3 deploys a novel scheme, unified modular ordering constraints (UMOC), to schedule FL/CL/RTL update blocks together under the same abstraction. UMOC is discussed in detail in Chapter 3. The intra-cycle ordering of RTL update blocks is implicitly inferred from the signals that each block reads or writes. The intra-cycle ordering of CL/FL update blocks is deduced from local explicit ordering constraints between method and/or update blocks, and the information of the methods each update block calls. The user can



(a) PyMTL3 Workflow



(b) PyMTL3 Framework

Figure 2.2: PyMTL3 Overview

simply set explicit ordering constraints in each component. The simulation passes will handle all the ordering constraints globally. UMOC eliminates the need to manually schedule CL update blocks to model the desired behavior and is the key mechanism in PyMTL3 to support seamless multi-level modeling. PyMTL3 simulation passes combine UMOC and Mamba++ (discussed in detail in Chapter 4) to provide high simulation performance.

Highly Parametrized Static Elaboration – Python's object-oriented programming and dynamic typing features enable PyMTL3 users to intuitively parametrize hardware components, as opposed to using low-level HDL's limited parametrization constructs and static typing. The users can use parameters of arbitrary types and instantiate different models or update blocks based on

```python
# Creating RTL register incrementer
# using PyMTL3 embedded DSL
class RegIncr( Component ):

 def construct( s, Type, inc=1 ):
  s.in_ = InPort ( Type )
  s.out = OutPort( Type )

  s.tmp = Wire( Type )
  @update_ff
  def seq_reg():
   s.tmp <<= s.in_

  @update
  def comb_out():
   s.out @= s.tmp + inc

class RegIncrNstage( Component ):

 def construct( s, Type=Bits32, N=1 ):
  s.in_ = InPort ( Type )
  s.out = OutPort( Type )

  s.rs = [ RegIncr( Type )  \
           for _ in range(N) ]

  connect( s.rs[0].in_,  s.in_ )
  connect( s.rs[-1].out, s.out )

  for i in range(N-1):
   # //= is syntactic sugar for connect
   s.rs[i].out //= s.rs[i+1].in_

# Parametrization using PyMTL3 embedded DSL
dut = RegIncrNstage( Bits16, 3 )
dut.set_param("top.rs[0].construct", inc=5 )
dut.set_param("top.rs[2].construct", inc=13)

# Static elaboration to create PyMTL3 NIMIR
dut.elaborate()

# Calling NIMIR API
print( dut.get_input_ports() )

# Apply PyMTL3 passes on the NIMIR model
dut.apply( RefactoringAnalysisPass() )
dut.apply( CheckInferedLatchPass() )

# Default pass group includes the UMOC graph
# generation pass, UMOC scheduling pass,
# and the simulation pass
# textwave=True enable textwave pass
dut.apply( DefaultPassGroup(textwave=True) )

# Call simulation method added by the
# simulation pass
dut.sim_reset()

dut.in_ @= 0
dut.sim_tick()

# Print text-based waveform
dut.print_textwave()
```

(a)

```python
# Creating FL checksum accelerator
# using PyMTL3 embedded DSL
class ChecksumXcelFL( Component ):

 def read( s, addr ):
  return s.reg_file[ int(addr) ]

 def write( s, addr, data ):
  s.reg_file[ int(addr) ] = b32(data)

  # If go bit is written
  if s.reg_file[4]:
   words = []
   for i in range( 4 ):
    words.append( s.reg_file[i][0 :16] )
    words.append( s.reg_file[i][16:32] )
   s.reg_file[5] = checksum( words )

 def construct( s ):
   # The FL accelerator minion interface is
   # hooked up directly to local methods
   s.xcel = XcelMinionIfcFL(read=s.read,
                            write=s.write)

   # Components
   s.reg_file = [ b32(0) \
                  for _ in range(6) ]

# Creating data-class like Pythonic
# high-level user-defined bitstruct types
def mk_xcel_req_msg( addr, data ):
  @bitstruct
  class XcelReqMsg:
    type_ : Bits1
    addr  : mk_bits( addr )
    data  : mk_bits( addr )
  return XcelReqMsg

# similar to mk_xcel_req
def mk_xcel_resp_msg( data ):
  ...

# Creating RTL processor
# using PyMTL3 embedded DSL
class ProcRTL( Component ):

 def construct( s ):
  s.xcel = XcelMasterIfcRTL( \
            mk_xcel_req_msg( 5, 32 )
            mk_xcel_resp_msg( 32 ) )
  ...

class TestHarness( Component ):
 def construct():
  s.proc = ProcRTL()
  s.xcel = ChecksumXcelFL()

  # Polymorphic interface connections
  connect( s.proc.xcel, s.xcel.xcel )
  ...
```

(b)

Figure 2.3: PyMTL3 Code Example

value or type. Moreover, PyMTL3 provides a *powerful parameter configuration system* to solve the common pitfall of parametrizing a hierarchical design. Usually the designer must pass the same parameter from the top-level design through the entire hierarchy. In PyMTL3, the designer can instead specify the parameter at the top-level component using a string with wildcard selection. PyMTL3 will resolve simple regular expressions and distribute the parameters accordingly. Lines 35–37 of Figure 2.3(a) show how the individual `RegIncr` components in the array are configured. In practice, this system can significantly reduce the chance of misconfiguration in a complex system-on-chip composed by many hardware generators.

Polymorphic Interface Connections – PyMTL3 interfaces are bundles of value ports or method ports. By default, connecting two interfaces involves recursively connecting nested interfaces and ports pairs with the same name. However, the designer may want to insert an adapter between two incompatible interfaces. In highly parametrized PyMTL3 design generators, manually inserting such adapters is tedious and error-prone due to the verbose type introspection code that checks for matching interface pairs and duplicated code across different components that instantiate the same interface pair. For example, composing any FL/CL/RTL components often involves inspecting the interface type and inserting the corresponding cross-level adapters. To solve this problem, PyMTL3 allows the interface designer to provide *a customized connect method* in the interface class to centralize type introspection and adapter insertion code. When connecting two interfaces, PyMTL3 *automatically* invokes the customized connect and falls back to by-name connection if no match is found. Lines 59 of Figure 2.3(b) show the connection of an FL interface (created in lines 5–23 of Figure 2.3(b)) and an RTL interface (instantiated in lines 48–50 of Figure 2.3(b)).

High-Level User-Defined Data Types – Inspired by Python's `dataclass`, PyMTL3 supports arbitrarily arrayed/nested user-defined data types for *both native-Python simulation and HDL generation*. PyMTL3 provides Pythonic `dataclass`-like APIs to declare new data types (lines 32–36 of Figure 2.3(b)). The simulation passes can determine the sensitivity of subfields to correctly schedule the simulation. The translation passes can directly generate nested SystemVerilog struct types, or recursively map subfields to slices of a flattened signal (for Verilog).

PyH2: Property-Based Random Testing – PyMTL3 includes PyH2, a property-based random testing framework for hardware generators, processors, and hardware data structures. PyMTL3 provides carefully implemented `hypothesis` composite search strategies to generate random Bits and user-defined type objects. One key advantage of PyH2 over traditional random testing and

iterative-deepened testing is that PyH2 first samples the test-case space *and design-parameter space* to quickly find a failing test case and then automatically shrinks the failing case *and the design parameters*. The result is a minimal failing case with minimal design parameters (e.g., shrinking a 50-transaction case for an eight-node network to a 10-transaction case for a four-node network). PyH2 is discussed in detail in Chapter 5.

2.3.2 PyMTL3 NIMIR and Elaboration

PyMTL3 implements the NIMIR architecture and exposes APIs for passes to invoke. PyMTL3 NIMIR mostly has dictionaries and lists as data structures to store the ordering constraints, the logic blocks, and their relationship. Then, PyMTL3 NIMIR provides an API called `elaborate()` for the user to specify a top-level model. During the elaboration process, PyMTL3 recursively collects all the metadata throughout the model hierarchy and stores the collected metadata at the top level model object. Most PyMTL3 NIMIR APIs are called from the top-level to return/modify the hierarchy such as retrieving all child modules that match a certain name filter, and retrieving all the logic blocks throughout the hierarchy as shown in Figure 2.2(b).

2.3.3 PyMTL3 Passes

PyMTL3 passes are systematic programs that interact with the PyMTL3 NIMIR. The categorization of PyMTL3 passes follow the NIMIR specification: analysis passes, instrumentation passes, and transform passes. Many PyMTL3 passes leverage open-source Python libraries and reuse/target open-source hardware tools, which confirms the benefits of using a powerful host language to build a hardware modeling framework.

Analysis Passes

Analysis passes should only query the metadata from the data structures stored in NIMIR. Hence they are used to traverse the model hierarchy and extract useful information for the designer to characterize the model.

Linting Passes – Linting passes check the coding style of PyMTL3 hardware descriptions. The `CheckSignalNamePass` enforces a naming convention on all the signals in the model and reports violations. It calls one of the API to query all of the signals in the hierarchy, and then

checks each signal's name against a given checker function, i.e., a Python lambda function that returns true/false. The `CheckUnusedSignalPass` report signals that are declared but never used. It calls APIs to query all of the signals, all of the update block read/write information, and all of the connections. It then uses the set data structure to figure out the unused signals.

Statistics Passes – Statistics passes are used to extract and/or visualize characteristics of the design. The `RefactoringAnalysisPass` gives insights into code refactoring by using `matplotlib` to create a scatter plot of the total input/output bitwidth of each module and a histogram plot of all the update block lengths. This is a good example of leveraging other Python packages to significantly simplify the plotting process. `DumpUDGPass` leverages `graphviz` to visualize the directed graph of all update blocks as vertices and all dependencies between these blocks as edges, which can be very useful for debugging unexpected cyclic dependencies.

Pre-Synthesis Passes – Pre-synthesis passes attempt to address RTL synthesis related issues. The `CheckInferredLatchPass` reports potential inferred latches by querying the AST of update blocks to check if each signal written in the block has valid assignments in all conditional branches. The `CheckClockGatingPass` reports all signals that are inferred to flip-flops, but non-blocking assignments are not included in an `if` statement block. Early-stage estimation passes give rough estimates of the hardware based on annotated area/power/timing without invoking external tools. The `AreaEstimationPass` reports the aggregated area from the annotated area estimates of all leaf components in a structurally composed design.

Instrumentation Passes

Instrumentation passes only adds functionality to the model, and should not change the hardware model itself. The added functionality can vary from an added simulator and corresponding APIs to perform cycle-by-cycle simulation, hooks and APIs to print out the internal states of the model, to attaching useful metadata to the model.

Simulation Passes – Building under the NIMIR concept, PyMTL3 natually becomes a platform for simulation mechanism research. Simulation passes are instrumentation passes that add simulating methods to the top-level component for the user to simulate the whole design. Each simulation pass implements different modeling semantics and/or creates a different simulator for different simulation performance. Researchers can add new simulation passes to explore new

scheduling mechanisms without modifying existing passes. The `EventDrivenPass` can schedule pure-RTL models with cyclic dependencies between update blocks and throw exceptions for actual combinational loops. The pass queries the read/write information of all update blocks and constructs sensitivity information to decide the dependent blocks of each update block. The added tick function maintains an event queue to trigger update blocks. The `StaticSchedulingPass` can only schedule models without cyclic dependencies even though they may not be actual combinational loops. However, removing the event queue leads to higher simulation performance when the toggle rate is high. The pass constructs a direct acyclic graph and applies a topological sort to compute a linear execution schedule for every cycle. The added tick function simply iterates over the static schedule. The `DynamicSchedulingPass` can schedule models with cyclic dependencies using the strongly connected component (SCC) algorithm. Our previous paper on Mamba [JIB18] proposed several novel scheduling techniques that boost the simulation performance under the `PyPy` just-in-time compiler *in a pure-Python environment*. The techniques are implemented as additional simulation passes. Note that the paper discusses the techniques for static scheduling, and I have successfully built a hierarchical static scheduling pass to optimize the simulation performance for any graph with cyclic dependencies. Details of the simulation techniques and the scheduling algorithms are discussed in Chapter 4. This confirms the extensiblity of NIMIR to sustain a research platform.

Tracing Passes – It is important for a productive hardware modeling framework to provide different tracing options to debug or visualize the execution. In PyMTL3, we have built many tracing passes to assist the designer. Tracing passes are instrumentation passes that add corresponding tracing hook functions to specific point of the scheduled execution. The hook functions captures the internal signal values. The classic `VcdGenerationPass` adds a callback function before the simulated rising clock edge to record the value changes. Simulations with this pass will provide a file in the VCD format compatible with GTKWave, an open-source waveform viewer. Inspired by PyRTL, the `TextWavePass` horizontally visualizes per-cycle value changes of every signal using ASCII text sequences after the execution. `VerilogTBGenPass` captures the cycle-by-cycle value change of the interface signals of a marked component, and generates a Verilog test bench with assertions for use in pure-Verilog four-state RTL or gate-level simulation. Note that the VerilogTBGenPass complements the PyMTL3 native testing with the ability to perform 4-state simulation

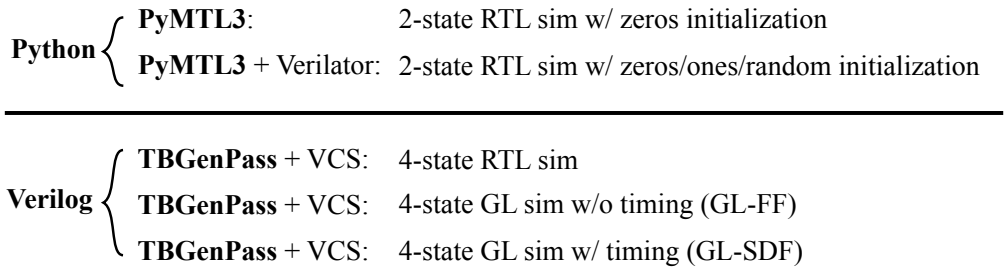


Figure 2.4: VerilogTBGenPass Completes the PyMTL3 Testing Spectrum – PyMTL3 native simulation and Verilator co-simulation can only perform 2-state simulation with different initialization options. The VerilogTBGenPass generates Verilog test harness based on the simulation in native Python, so that the generated Verilog can be simulated in Synopsys VCS using 4-state simulation.

in Synopsys VCS as shown in Figure 2.4, which drastically improves PyMTL3's interoperability with ASIC flows.

Translation Passes – Another type of useful instrumentation passes are translation passes which attach the translated IR and/or source file to the design. These passes are also a key part of the two-level IR structure as mentioned in Section 2.2. In PyMTL3, we build HDL translation passes so that the designer can translate PyMTL3 RTL code into HDL code that is compatible with open-source/commercial FPGA/ASIC synthesis tools. The `RTLIRGenPass` first lowers the RTL design from NIMIR into RTLIR, a low-level hardware IR provided by PyMTL3. Then the translation backend pass turns the RTLIR into corresponding HDL source code. Having the RTLIR as the input to different translation backends and implementing backends as passes already streamlines the process of adding a new backend. Moreover, PyMTL3 ships a carefully designed translation framework that provides a code generator template to be specialized by the target HDL backend with the mapping from RTLIR primitives to HDL source code. In other words, the user only needs to fill in the blanks to add a new backend. A backend can also inherit from an existing backend to maximize code reuse. For example, the Yosys-SystemVerilog backend inherits most code generation functions from the regular SystemVerilog backend and only adds several hundred lines of code to override the interface/struct-specific functions.

Transform Passes

Transform passes systematically modify the hardware model itself using at runtime using NIMIR APIs, which opens up various opportunities to avoid making massive temporary modifications and reversions to the design codebase.

Import Passes – PyMTL3 provides import passes to integrate external IPs with PyMTL3 designs/testbenches using black-box import (simulation only) or white-box import (creating a new PyMTL3 component with internal constructs). Co-simulating existing IPs in Python significantly facilitates verification. Import passes are transform passes that create PyMTL3 components on-the-fly and replace the original placeholders so that the external IPs are integrated seamlessly with rest of the design hierarchy. SystemVerilog and SystemC IPs are imported as black-box modules backed by external C++ shared libraries. The user needs to specify interfaces and source files in the placeholder. Specifically, the `VerilogImportPass` leverages Verilator to generate a C++ simulator for all specified SystemVerilog files, generates a C interface wrapper, and links the C++ simulator against the wrapper to produce a C++ shared library. Similarly, the `SystemCImportPass` directly creates a C++ shared library by compiling a generated C++ interface wrapper with the SystemC code and the SystemC kernel library. Then, the placeholder is replaced by a generated PyMTL3 wrapper component that communicates with the shared library through Python's C foreign function interface.

Prototype Proxy Passes – After pushing the RTL model through an FPGA/ASIC flow, PyMTL3 provides prototype proxy passes that integrate the real prototype with *the same Python test bench*, which can significantly improve the prototype testing productivity compared to an ad-hoc flow. The proxy passes extensively use Python reflection and NIMIR APIs to generate wrapper components that wrap around the prototype. The PyMTL3 test bench can send data to the wrapped prototype over the same interface as the original RTL model, as the wrapper components will serialize/deserialize the data and communicate with the system device.

Ad-Hoc Transform Passes – Motivated by real-world situations, PyMTL3 provides many ad-hoc transform passes to help avoid making significant modifications (that may be reverted eventually) to the codebase. These passes creatively exploit the `add`, `delete`, and `replace` APIs to mutate the design hierarchy in-situ and open up many opportunities for productive verification and rapid prototyping that would be challenging in other frameworks. Leveraging Python's dynamic typing feature, the `AddDebugSignalPass` pulls a signal from deep in the hierarchy to expose it at the top level for debugging. For example, the pass takes a signal's hierarchical name `top.chip.tiles[1].core.dpath.mult.en`, iteratively inserts a `debug_en` port to the multiplier, the datapath, the core, the tile, the chip, and the top, and connects all of the added ports together. The user can then apply translation passes to generate HDL code with the additional

ports. `SwapHardenedIPPass` searches for instances of marked PyMTL3 behavioral models and swaps them with placeholders that import hardened Verilog models. Co-simulating the design with real hardened models improves the fidelity of the tests.

2.4 Developer's Case Study: Supporting Delay-Annotated Gate-Level Modeling

In this section, I present a developer's case study to demonstrate the extensibility of the NIMIR architecture in the PyMTL3 framework. I illustrate how to support delay-based gate-level modeling on top of existing RTL and CL modeling but without affecting the existing code base. The case study is based on official release version of PyMTL3. Figure 2.5 shows the envisioned PyMTL3 design code of a positive-edge-triggered D-latch model and the D flip-flop model. After the three steps illustrated below to enhance PyMTL3 eDSL, PyMTL3 NIMIR, and PyMTL3 passes, PyMTL3 should be able to support this code.

2.4.1 Adding Embedded DSL Primitives

1. To keep the added GL modeling primitives separate from existing code base, we implement a new class `GLComponent` inherited from `ComponentLevel7` as shown in Figure 2.6(a), so that all the RTL modeling primitives such as `update` and `update_ff` can directly be reused. PyMTL3 EDSL modeling primitives are implemented in the `pymtl3.dsl` package. Different component levels are used internally incrementally add support for modeling primitives.
2. Then we override the `__new__` method to add the function-to-delay mapping dictionary `upblk_delay` to the private namespace `s._dsl` in lines 1–5 of `upblk_delay`. PyMTL3 NIMIR stores all the metadata in this `s._dsl` namespace. Note that line 2 of Figure 2.6(a) invokves the the parent class `__new__` method as a convention since we still want to leverage previously implemented primitives.
3. We add the `update_delay(func, delay)` construction-time modeling API to the component so that the user can mark a function as an update block with a specific delay as shown in line 7–12 of Figure 2.6(b). Inside `update_delay(func, delay)`, it stores the function and the delay to the mapping dictionary at the host component where the update block is

```
class PosTrigDLatch( GLComponent ):
  # Parametrized by the delay in nano second
  def construct( s, delay ):

    # input clock signal from clock generator
    s.in_clk = InPort()

    s.D = InPort()
    s.Q = OutPort()

    # We want to use @update_delay decorator to mark the delay
    # of an update block.
    # We want to use "|=" (bar-equal) operator for delayed assignments
    @update_delay(delay)
    def update_dlatch():
      s.Q |= s.D if s.in_clk else s.Q

class DFF( GLComponent ):
  def construct( s ):
    s.in_clk = InPort()
    s.D = InPort()
    s.Q = OutPort()

    s.DL1 = PosTrigDLatch( delay=50 )
    s.DL2 = PosTrigDLatch( delay=50 )

    s.DL1.in_clk //= lambda: ~s.in_clk
    s.DL2.in_clk //= lambda: s.in_clk
    s.D //= s.DL1.D
    s.DL1.Q //= s.DL2.D
    s.DL2.Q //= s.Q

x = DFF()
x.elaborate()
x.apply( GenDAGPass() )
x.apply( EventSchedulePass() )

x.in_clk @= 0
x.D @= 1
x.sim_delay(1000)
x.in_clk @= 1
...
```

Figure 2.5: Example Design for Delay-Annotated Gate-Level Modeling – The design is a positive-edge-triggered D-latch and a D flip-flop that composes two of the latches. We want to use delay annotation on update blocks to model delayed logic while still supporting zero-delay combinational logic.

created. Then it invokes the `_cache_func_meta` API as the convention to cache the AST of the function with the `|=` operator.

4. Outside the class, we add an `@update_delay(delay)` decorator as syntactic sugar for the user to succinctly mark the delay-annotated blocks. The nested function implementation in Figure 2.6(c) is the most Pythonic way to create a decorator with a decorator parameter. The

```python
from pymtl3.dsl.ComponentLevel7 import ComponentLevel7

class GLComponent( ComponentLevel7 ):
  ...
```

(a)

```python
  def __new__( cls, *args, **kwargs ):
    inst = super().__new__( cls, *args, **kwargs )

    inst._dsl.upblk_delay = {}
    return inst

  def _update_delay( s, blk, delay ):
    ComponentLevel1._update( s, blk )

    s._dsl.upblk_delay[ blk ] = delay

    s._cache_func_meta( blk, 4, ast.BitOr )
```

(b)

```python
# the @update_delay decorator implementation
def update_delay( delay ):
  def real_decorator( blk ):
    NamedObject._elaborate_stack[-1].update_delay( blk, delay )
    return blk
  return real_decorator
```

(c)

```python
class Bits:
  ...

  def __ior__( self, v ):
    nbits = self._nbits

    ... # type checks

    try:
      self._nexts.append( _next )
    except AttributeError:
      self._nexts = deque( [ _next ] )

    return self

  def _advance( self ):
    try:
      self._uint = self._nexts.popleft()
    except Exception:
      pass
```

(d)

Figure 2.6: PyMTL3 EDSL Implementation to Support Delay-Annotated GL Modeling – (a) shows the new GLComponent class; (b) overrides `__new__` method to add data structures without the need for user to manually override `__init__` and the private method to add new data to the data structure; (c) is the decorator function implementation that leverages Python mechanisms; and (d) shows the Bits enhancement to support `|=` delayed assignment including a new operator and a method.

decorator finds the latest component in the global elaboration stack and invokes the previous `update_delay(func, delay)` method on the component.

5. Since we want to use a new operator `|=` on signals for delayed assignment, we add the `__ior__` operator to the datatype `Bits` class that contains a list of delayed assignment values. This is necessary since it is possible to have multiple buffered values for the same signal at different future timestamps. We also add the `_advance()` API to the Bits object to use the next buffered value as the signal value.

As shown above, we follow the convention of existing APIs, and add merely tens of lines of code to support the new DSL modeling primitive. It is worth noting that previous design code is not affect by the added GLComponent class and the added primitives at all. This confirms the modularity of the NIMIR architecture.

2.4.2 Adding NIMIR Data Structures and APIs

1. The PyMTL3 NIMIR elaboration process basically collects all the metadata from all the child components and centralizes them in the top level component on which the `elaborate()` method is called. The PyMTL3 NIMIR implementation provides flexible sub-methods of the elaboration process for inherited classes to override. This avoids the need to modify existing code to add new features. As shown in lines 1–4 of Figure 2.7(a), we simply override the private `_elaborate_declare_vars` method, use `super()` to call the method in the parent class, and declare the `all_upblk_delay` dictionary to store the mapping of all the delay-annotated update blocks and their corresponding delays. Because Python functions are unique objects, we do not need to worry about duplicate keys in the dictionary.

2. Similarly, we override the `_collect_vars` method to add the desired behavior during the data collection process as shown in lines 6–10 of Figure 2.7(a). This method is supposed to be called on the top level and has a parameter `m`, which is the child component to collect. Hence the desired behavior is simply merging the local `upblk_delay` dictionary of the child component into the global `all_upblk_delay` dictionary.

3. Finally, we add NIMIR APIs to expose the newly added global delay dictionary. To expose the whole dictionary, We simply add a `get_all_update_delay()` method to the component. Lines 1–6 of Figure 2.7(b) shows the implementation. It starts with a check function

```python
# Override
def _elaborate_declare_vars( s ):
  super()._elaborate_declare_vars()
  s._dsl.all_upblk_delay = {}

# Override
def _collect_vars( s, m ):
  super()._collect_vars( m )
  if isinstance( m, GLComponent ):
    s._dsl.all_upblk_delay.update( m._dsl.upblk_delay )
```

(a)

```python
def get_all_update_delay( s ):
  try:
    s._check_called_at_elaborate_top( "get_all_update_delay" )
    return s._dsl.all_upblk_delay
  except AttributeError:
    raise NotElaboratedError()

def get_delay_of_update_block( s, blk ):
  try:
    s._check_called_at_elaborate_top( "get_delay_of_update_block" )

    assert blk in s._dsl.all_update_delay, \
           f"{blk} is not annotated with delay!"

    return s._dsl.all_update_delay[ blk ]

  except AttributeError:
    raise NotElaboratedError()
```

(b)

Figure 2.7: PyMTL3 NIMIR Implementation to Support Delay-Annotated GL Modeling – (a) shows the implementation to override elaboration steps to collect delayed update blocks; and (b) shows the APIs that expose the collected metadata.

that checks if the API call is performed on an elaborated component, and then directly returns the `all_update_delay` dictionary created during elaboration. Also, as shown in Lines 8–18 of Figure 2.7(b), we can add another API called `get_delay_of_update_block(blk)` for passes that already query all update blocks to get the delay of a specific update block.

In summary, we only need to add 10 lines of Python code in PyMTL3 NIMIR implementation to enhance the elaboration process and 20 lines of code to add two APIs leveraging many existing utility functions. This further confirms the flexibility and extensibility of the NIMIR architecture.

2.4.3 Adding Event-Driven Scheduling Passes

After adding EDSL primitives and NIMIR APIs, we need to develop the event-driven scheduling passes to support delay-annotated GL simulation. Existing simulation passes are cycle-based

```
1  assert not top.get_all_update_ff()
2  assert not top.get_all_update_once()
3
4  all_upblk_delay_dict = top.get_all_update_delay()
5  all_upblk_reads_dict, all_upblk_writes_dict, _ = \
6    top.get_all_upblk_metadata()
7
8  V = top._dag.final_upblks
9
10 top._sched.preamble = preamble = []
11
12 # Preprocessing preambles
13 for b, reads in all_upblk_reads_dict.items() | \
14                 top._dag.genblk_reads.items():
15   delay = all_upblk_delay_dict.get( b, 0 )
16   for r in reads:
17     if r.is_input_value_port() and r.is_top_level_signal() and \
18        r.get_host_component() is top:
19       preamble.append( ( delay, b ) )
20
21 top._sched.triggers = triggers = { v: [] for v in V }
22
23 # Preprocessing triggered events for delayed assignments
24 for b, writes in top._dsl.all_upblk_writes.items():
25   if b in all_upblk_delay_dict:
26     delay = all_upblk_delay_dict[b]
27     for w in writes:
28       triggers[b].append( ( Event.ADVANCE, delay, signal_advance_dict[w]) )
29
30 # Preprocessing triggered events for subsequente update blocks
31 for (u, v) in top._dag.all_constraints: # u -> v
32   if u in V and v in V:
33     delay = all_upblk_delay_dict.get( v, 0 )
34     triggers[u].append( ( Event.TRIGGER, delay, v ) )
```

Figure 2.8: Preprocessing NIMIR Metadata For Event-Driven Scheduling – This part of the event-driven scheduling pass first calls APIs to get all the update blocksand the delays. Then it executes a few nested loops to establish the triggering relationships and corresponding signal advance events. The preambles are events triggered by top-level input value changes.

and cannot be directly reused, but we are able to reuse some of the previous passes such as the UDG generation pass to generate sensitivity information of update blocks.

1. First, we need to invoke a few NIMIR APIs to obtain the metadata. Lines 1–8 of Figure 2.9 invokes sevaral APIs to perform checks and to obtain read/write metadata, and retrieves all the update blocks from the results of UDG generation pass. Lines 10–19 prepares the preamble events that propagate all the modification to input signals outside the simulator such as lines 42–43 of Figure 2.5. Line 16–18 enumerates all the signals that an update block reads and performs more NIMIR API calls on the signals to see if any of the signals are top-level input ports. Lines 23–28 prepares the triggered assignment events of all `update_delay` blocks. This is because the value change of `|=` assignments inside an `update_delay` block must

```
1  top._sched.event_queue = []
2  top._sched.timestamp   = 0
3
4  def create_sim_delay( top ):
5    event_queue = top._sched.event_queue
6    preamble    = top._sched.preamble
7    triggers    = top._sched.triggers
8
9    def sim_delay( delay ):
10     time = top._sched.timestamp
11     target_time = time + delay
12
13     # Check if top-level ports are written using @=
14     top._check_top_level_inports()
15
16     # execute preamble blocks that read input ports
17
18     for delay, event in preamble:
19       event()
20       triggered_time = time + delay
21       for p, t, e in triggers[event]:
22         heappush( event_queue, ( triggered_time, p, t, e ) )
23
24     while event_queue:
25       time, event_type, event_delay, event = event_queue[0]
26       if time > target_time:
27         break
28       heappop( event_queue )
29
30       event()
31
32       if event_type == Event.TRIGGER:
33         triggered_time = time + event_delay
34         for p, t, e in triggers[event]:
35           heappush( event_queue, ( triggered_time, p, t, e ) )
36
37     top._sched.timestamp = target_time
38
39   return sim_delay
40
41 top.sim_delay = create_sim_delay( top )
```

Figure 2.9: Event-Driven Scheduling Implementation for Delay-Annotated GL Models – The `sim_delay` function is created for each elaborated top. It creates a priority queue indexed by timestamps to capture the events. For each invocation of the `sim_delay` function, it first pushes all the preamble events to the event queue. It then iteratively execute them and trigger new events until the event queue is empty.

happen after the delay. In other words, we need to push the delayed assignment to the event queue as a triggered event. Lines 30–36 prepares the triggered subsequent update blocks. The `preamble` and `triggers` are they data structures used by the event-driven simulation.

2. Then we create the `sim_delay` function which simulates the design for a certain amount of time (and also takes the value changes of the top-level input ports into account). Lines 4–9 in Figure 2.9 shows how we use a nested function closure to capture the top in the generated `sim_delay` function. The `sim_delay` function takes an integer delay and simulates

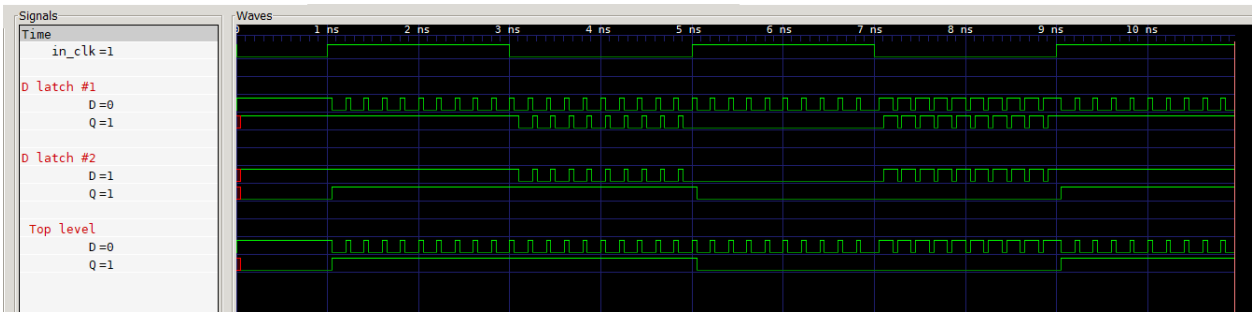


Figure 2.10: GTKWave Screenshot of the D Flip-Flop Simulation – This screenshot shows an example simulation of a changing input stimulus The two D latches show the expected behavior.

to the target time which is `timestamp + delay`. Similar to the existing simulation passes, we check if the top-level ports are written in a valid way in line 14. We execute all the pre-processed preamble blocks that read input ports to propagate the stimulus, and trigger subsequent events in line 24. Note that we use a priority queue (heap) whose key is a timestamp as the event queue. Then we pop events, execute them, and trigger more events until either the event queue is empty, or the timestamp of head of the priority queue is already larger than the target timestamp, as shown in lines 24–37.

3. Finally we enhance the scheduling tick to add hook for dumping the value changes into a `.vcd` file. This involves creating a vcd dumping hook function and invoke it near line 23 for preambles and line 35 of Figure 2.9. We also need to add an else branch to the if statement at line 32 and invoke the hook function, because the other type of events for delayed value updates will change the value of the signals and should be recorded instantly. We omit the implementation of the vcd hook function in this thesis because it is mostly details to deal with the VCD format and file input/output.

Figure 2.10 shows the `gtkwave` screenshot of one simulation run of the D flip-flop model in Figure 2.5. The stimulus is programmed to switch rapidly to exercise the latch behavior and the setup/hold time for the output. We can see the Q output of the first D-latch holds the value correctly when the clock is high, and changes with the top-level D input after a small delay correctly when clock is low. The Q output of the second D-latch holds the value correctly when the clock is low, and changes when the clock is high. Overall, the D flip-flop behavior is correctly simulated by the event-driven scheduling function generated by the PyMTL3 pass.

2.5 PyMTL3 for Open-Source Hardware

PyMTL3 is an ideal framework to jump-start the open-source hardware ecosystem for three major reasons:

- *PyMTL3 is embedded in Python.* Python is currently the most popular programming language for its high productivity. Python has been evolving for nearly three decades, supported by a large *open-source* community with over 100,000 third-party libraries. PyMTL3 users can use these third-party libraries to build test benches, golden reference models, and passes. For example, PyMTL3 analysis passes can leverage `matplotlib` and `graphviz` to visualize characteristics of hardware designs. Open-source hardware built in PyMTL3 can also directly reuse Python's package-management system `pip` for distribution. For example, installing PyOCN [TOJ+19] (an open-source on-chip network generator built with PyMTL3) involves a single command (`pip install pymtl3-net`), during which `pymtl3` and other dependencies are automatically installed.
- *PyMTL3 emphasizes interoperability with other open-source hardware tools.* A significant amount of open-source hardware is written in Verilog or SystemVerilog. Verilator is currently the fastest and most capable open-source simulator for synthesizable Verilog and SystemVerilog. Unfortunately, Verilator requires driving these simulations with low-level C++. PyMTL3 passes can automatically use Verilator to import Verilog and SystemVerilog models into PyMTL3 for black-box co-simulation. This enables PyMTL3 to combine the familiarity of Verilog/SystemVerilog with the productivity of Python. PyMTL3 passes can also support black-box co-simulation with SystemC, translate RTL models to Yosys-compatible or Verilator-compatible SystemVerilog, and generate GTKWave-compatible waveforms. We have also implemented a FIRRTL [IKL+17] backend that generates PyMTL3 model.
- *PyMTL3 promotes agile and test-driven design methodologies.* PyMTL3 adopts `pytest`, a mature full-featured Python testing tool to collect, manage, parametrize, and refactor tests. PyMTL3 also includes the PyH2 framework that repurposes `hypothesis`, a property-based testing (PBT) framework for Python software, to test hardware generators (PyH2G), processors (PyH2P), and hardware data structures (PyH2O). Currently, there is no standard verification methodology for open-source hardware. Open-source simulators (e.g., Verilator and Icarus Verilog) have limited support for industry standard verification methodologies (e.g.,

UVM). `cocotb` embeds Python in a Verilog simulator, which can limit the use of Python features. PyMTL3 takes the opposite approach by embedding Verilog in Python using Verilator, which unleashes the full potential of the Python runtime. Additionally, `cocotb` only targets building test benches, while PyMTL3 is a full-fledged modeling framework. Combining the familiarity of Verilog/SystemVerilog with the productivity features of Python, PyMTL3 realizes the agile hardware manifesto [LWC+16].

2.6 Conclusion

In this chapter, I proposed native in-memory intermediate representation (NIMIR), a novel and systematic approach to build extremely flexible and extensible hardware generation and simulation frameworks. I also presented PyMTL3, the first HGSF ever built using the NIMIR approach. PyMTL3 takes advantage of the existing Python ecosystem, emphasizes interoperability with other open-source tools, and provides strong support for agile test-driven design. Moreover, the flexible, modular, and extensible software architecture enables the PyMTL3 framework itself to evolve alongside the open-source hardware ecosystem. PyMTL3 has been open-sourced at `https://github.com/pymtl`.

CHAPTER 3
UMOC: UNIFIED MODULAR ORDERING CONSTRAINTS TO UNIFY CL AND RTL MODELING

The second key challenge in modern hardware modeling frameworks as mentioned in Section 1.2 is *the absence of a unified cycle-level and RTL modeling abstraction*. This essentially leads to fragmentation in the computer architecture community in terms of CL/RTL modeling methodology. A unified CL/RTL modeling mechanism can potentially build a bridge between computer architects who extensively model hardware in CL simulators and computer engineers who extensively implement hardware in RTL.

In this chapter, I propose unified modular ordering constraints (UMOC), a novel approach that seamlessly unifies method-based cycle-level (CL) modeling and signal-based register-transfer-level (RTL) modeling, to address the modeling abstraction challenge. Motivated by the challenges in state-of-the-art CL modeling methodologies and existing CL/RTL composition attempts, UMOC successfully breaks the trade-off between model fidelity and scheduling modularity for CL modeling and provides seamless composition of CL and RTL models. Instead of requiring the designer to specify the global intra-cycle ordering of hardware processes, UMOC eliminates this burden using implicit local ordering constraints of RTL signals and explicit local ordering constraints of CL methods. UMOC has been implemented and evaluated in PyMTL3, and has become the key modeling mechanism of PyMTL3.

3.1 Introduction

In response to the growing register-transfer-level (RTL) design effort for modern systems-on-chips (SoC) and the increasing heterogeneity in these SoCs, computer architects have been leveraging domain-specific cycle-level (CL) simulators (CPU [You07, BBB+11, PACG11], memories [RCBJ11], GPU [BYF+09, SBM+19], and on-chip networks [AKPJ09, LSC+10, boo11]), and general-purpose CL modeling frameworks [GTBS13, Pan01, LZB14], to facilitate early design-space exploration. Even though CL models include less hardware detail and usually cannot be converted to hardware, the faster simulation speed and easier modification/enhancement is crucial to the early design-space exploration phase. The approximate timing behaviors, combined with analytical area/energy/timing models [LAS+09], provide valuable insights to help make first-

order design decisions and hence drastically reduce the time spent later in the RTL development phase. After the CL design-space exploration phase, instead of moving directly from a complete CL model to a complete RTL implementation, the ability to seamlessly mix and match RTL models with CL models brings significant productivity benefits. Gradually swapping CL blocks for newly developed RTL blocks makes it easier to: (1) maintain the integration tests, end-to-end tests, and performance regressions, and (2) steadily improve the model fidelity of the whole design. Prior research attempts to unify the cycle-level descriptions and RTL generation for specific hardware domains (e.g., architectural description languages for processors [HGG+99, CML08]). *This work focuses on general-purpose CL/RTL modeling and composition mechanisms.*

Unlike RTL modeling's well-established discrete-event simulation semantics, the inter-cycle and intra-cycle semantics are different across different CL simulators. Commonly used CL inter-cycle mechanisms include: (1) discrete-event simulation that maintains an event queue to automatically advance the timestamp and trigger designer-scheduled events of hardware processes [You07, AKPJ09, BBB+11, Pan01], and (2) cycle-by-cycle simulation which essentially assumes all hardware processes are recurringly triggered at every rising clock edge [BYF+09, RCBJ11, JBM+13, GTBS13, LZB14]. When several hardware processes are triggered at the same timestamp in both cases, the intra-cycle mechanism has to decide the order of execution. *This work focuses on intra-cycle mechanisms.* The most commonly used CL intra-cycle mechanism is designer-specified global ordering of hardware process invocations for modeling combinational/sequential behaviors. However, global intra-cycle ordering makes it challenging to achieve model fidelity *and* scheduling modularity at the same time. State-of-the-art mechanisms for composing CL and RTL models are ad-hoc and only enable heterogenous compositions across different models of computation, due to the intra-cycle semantic gap between CL and RTL modeling. As elaborated in Section 3.2, we identify two major challenges in state-of-the-art CL simulators/frameworks and attempts to compose CL and RTL models: (1) the trade-off between model fidelity and scheduling modularity in CL modeling; (2) seamless composition of CL and RTL models.

In this chapter, I introduce a novel intra-cycle modeling mechanism that unifies method-based CL modeling and signal-based RTL modeling to solve these challenges. Unified modular ordering constraints (UMOC) provide a unified view for general-purpose CL and RTL modeling and enable automatically scheduling all the CL/RTL processes with designer-specified (CL) or inferred (RTL) *local* constraints without manually specified global intra-cycle ordering of hardware processes.

Section 3.3 discusses the key idea and foundation of UMOC. UMOC can be implemented in any unified CL/RTL modeling framework (e.g., SystemC [Pan01]). This chapter will leverage the UMOC implementation in PyMTL3 [JPOB20] as an example implementation to explain UMOC in Section 3.4. See Chapter 2 for background on PyMTL3. Section 3.5 includes two case studies on how UMOC with PyMTL3 enables accurately composing CL/RTL processors and CL/RTL checksum accelerators, and a bigger CL/RTL manycore system.

This work makes the following contributions: (1) we identify two key challenges to CL modeling and CL/RTL composition; (2) we propose unified modular ordering constraints (UMOC) to address these challenges; and (3) we showcase the implementation of UMOC in PyMTL3 from necessary primitives to scheduling algorithms.

3.2 Related Work and Motivation

In this section, we identify two key challenges to CL modeling and CL/RTL composition, along with the corresponding related work.

Challenge #1: Trade-off between model fidelity and scheduling modularity in cycle-level modeling – Cycle-level simulators [You07, BBB+11, RCBJ11, BYF+09, AKPJ09] usually improve the model fidelity against the target architecture by specifying the intra-cycle total ordering of calling hardware processes to model the desired pipeline/combinational behavior. Figure 3.1(b–c) shows an example of a C++ simulator modeling the processor and the accelerator composition in Figure 3.1(a) using reversed invocation order for pipeline behavior. Note that invoking processor and accelerator schedules as blackboxes at the top level as shown in Figure 3.1(d) harms the model fidelity regardless of the invocation order of `proc.tick()` and `accel.tick()`. Essentially, simply composing two modular "pipelines" and concatenating their execution schedule gives up the possibility to interleave hardware processes in these pipelines and can create a behavior mismatch against the target architecture. This is a module-level cyclic inter-dependency that the modular tick approach cannot break. Admittedly, the designer should be able to break the modularity to improve performance fidelity as illustrated in Figure 3.1(e) to resolve the module-level dependency. However, to the best of our knowledge, we have rarely seen any simulator that abandons scheduling modularity, simply because it is hard to maintain a flattened top-level schedule of a complex hardware block (see Figure 3.1(f)), especially during incremental development. gem5 [BBB+11]

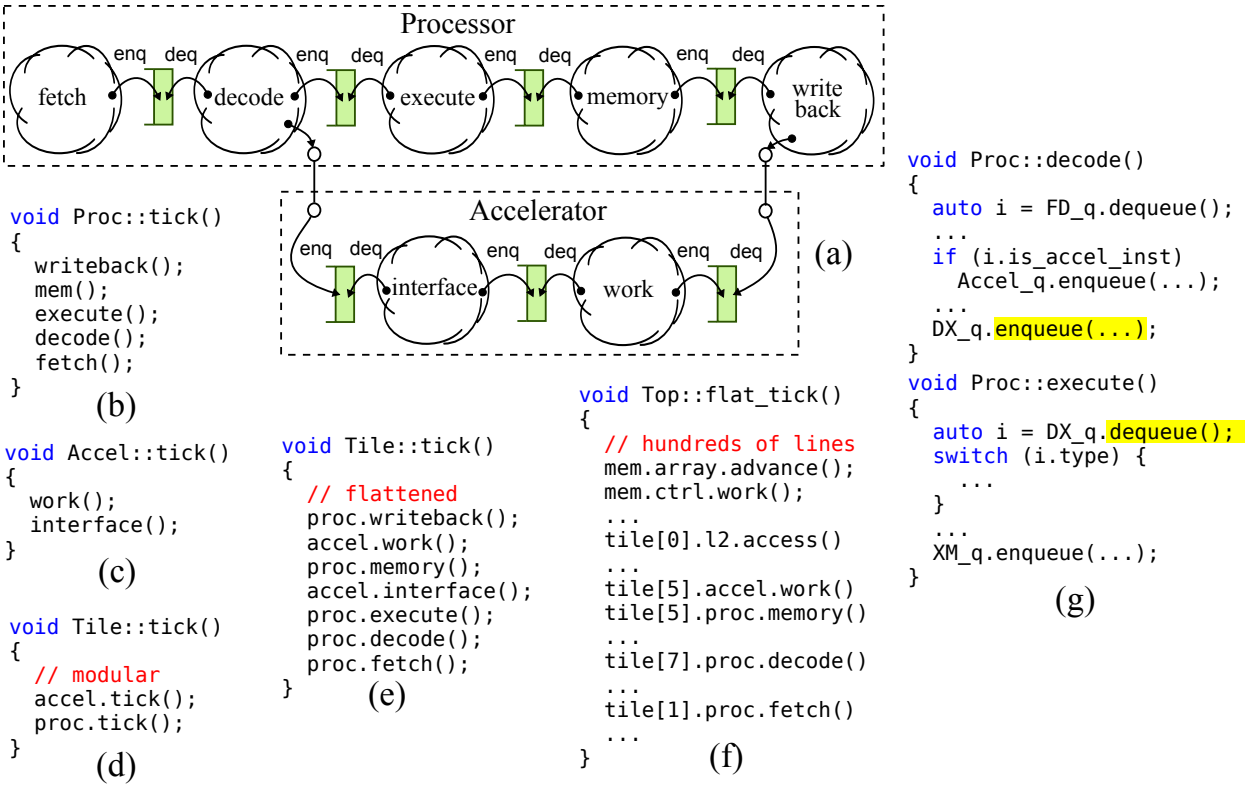


Figure 3.1: Modeling a Cycle-Level Processor/Accelerator Tile – An example abstracted from real-world simulator code: (a) the pipeline structure and composition of a five-stage processor and a two-stage tightly coupled accelerator where the accelerator request is sent out at `decode` and the response is accepted at `writeback`; (b–c) the tick methods of `Proc` class and `Accel` class, both of which model pipeline behavior; (d) the modular tick method of `Tile` that calls the tick of `Proc` and `Accel`; (e) the flat tick method that directly calls the hardware logic inside `Proc` and `Accel` for more accurate performance modeling; (f) the hypothetical flat tick function of a complex design that models the performance accurately; (g) `Proc::decode` and `Proc::execute` communicate through buffer `DX_q`.

relies on a designer-marked single-integer priority on each hardware process and decides the global intra-cycle ordering by sorting the events based on priority. Specifying incorrect priority will lead to unexpected and profound performance bugs such as erroneous combinational behavior between two decoupled modules, and it is impossible to report any mistake during scheduling under this scheme.

We conclude that the state-of-the-art CL modeling approaches rely on designer-specified *global* intra-cycle ordering of hardware processes, which makes it challenging to attain scheduling modularity *and* performance fidelity at the same time.

Challenge #2: Seamless composition of CL and RTL models – Several general-purpose modeling frameworks have provided first-class support for composing cycle-level models and RTL models. Cascade [GTBS13] is a CL modeling framework which provides RTL-like register elements and combinational updates as modeling primitives. Cascade supports composing

cycle-level models written in C++ with Verilog by exporting the CL model as a standalone C module and importing it inside a Verilog module using Verilog Procedural Interface (VPI). However, the top-level simulation driver is the Verilog simulator. SystemC [Pan01] provides a unified environment in C++ for CL and RTL modeling. However, SystemC primitives for transaction-level modeling are often used for functional verification rather than detailed performance modeling. The "transactors" [KTMH07] between TLM and RTL have to contain sequential elements which makes fine-grained intra-cycle CL/RTL composition difficult. In other words, it is impossible to model intra-cycle behavior going through RTL–CL–RTL if TLM channels are used as interfaces. PyMTL [LZB14] also unifies CL/RTL modeling in Python by instantiating port-based RTL interfaces inside CL models and wrapping RTL interfaces with CL buffers with enqueue/dequeue methods for CL processes to call. PyMTL supports event-driven semantics for RTL models, but the designer has to manually call the CL processes in a total order like Figure 3.1(b-f). Hence, PyMTL fails to close the CL/RTL semantic gap.

There are also ad-hoc attempts to compose established CL/RTL simulators. PAAS [LFSZ17] supports coarse-grained composition of Verilog RTL accelerators with gem5 CPU and memory models using linux `/dev/shm` shared memory to exchange data between gem5 and a Verilator-compiled [ver21] C++ simulator. Another attempt [GALP18] composes gem5's system simulation with the C++ library compiled from Chisel-generated Verilog code also using Verilator. MosaicSim [MMG+20] deploys an interleaver at the top level for scheduling events from CL and RTL tiles, but the RTL tile model only provides performance estimates instead of simulating real RTL code.

We conclude that previous attempts to compose CL and RTL models are ad-hoc and design-specific at a coarse granularity. As far as we are aware, no prior work has provided a seamless composition of CL and RTL models using a unified model of computation.

3.3 Unified Modular Ordering Constraints

In this section, I describe unified modular ordering constraints (UMOC), a novel *intra-cycle scheduling mechanism* to unify CL/RTL modeling which tackles the two challenges in Section 3.2. UMOC is an *intra-cycle scheduling mechanism*. and could be combined with either discrete-event simulation or cycle-by-cycle simulation. In state-of-the-art RTL simulators, the RTL processes

are automatically collected and scheduled according to event-driven execution semantics, which means that the designer is *unaware of the actual scheduling process*. However, state-of-the-art CL simulators usually requires the designer to *manually schedule CL processes* for desired timing behavior. Inspired by this difference, UMOC introduces explicit local ordering constraints between CL methods to *let the underlying scheduler automatically schedule the CL processes*. A unified directed graph is built from all CL/RTL processes and implicit/explicit ordering constraints to enable seamless intra-cycle composition of CL and RTL models. I also discuss how to handle cycles in the unified directed graph and how to schedule intra-cycle simulation.

3.3.1 RTL Scheduling with Implicit Constraints

If behavioral RTL process A writes signal x and B reads x, traditional HDL simulators will infer this sensitivity and dynamically schedule B to execute whenever A modifies x. Inspired by previous work on statically scheduling RTL processes [PMT04, GTBS13, JIB18], I propose to use the notion of ordering constraints to implicitly deduce the relationship between block A and B as follows.

$$\left.\begin{array}{c} x \text{ is a combinational wire} \\ A \text{ writes signal } x \\ B \text{ reads signal } x \end{array}\right\} \Longrightarrow \begin{array}{c} A \text{ precedes } B \\ (A < B) \end{array}$$

The key observation here is that even though x is merely a *local* variable w.r.t. A and B, the ordering between A and B is later used by the scheduler *globally* to determine the final execution order of all RTL processes in the design. This is because in a hierarchical RTL model, an RTL module exposes ports to the parent module which are connected to signals in other modules. All the connected signals are essentially the same signal, and hence the preceding relationship of any two faraway combinational RTL processes can be established without exposing any details inside the module, which preserves the *modularity*.

3.3.2 CL Scheduling with Explicit Constraints

For CL modeling, we also want to reduce the burden on designers by propagating local ordering constraints. However, there is no signal in CL models, as CL models manipulate high-level data

structures. We observe that CL processes still need to communicate via buffers that expose *methods* for CL processes to call (similar to SystemC `sc_fifo`). For example, Figure 3.1(g) shows that `decode` enqueues a message to `DX_q` and `execute` dequeues the message (using a queue handles the back pressure from a later pipeline stage). The reversed order in Figure 3.1(b) guarantees that `execute` is called before `decode` in every clock cycle, which means *`dequeue` of the buffer is always called before `enqueue`*. Thus, whatever `decode` enqueues to the buffer will only be dequeued by `execute` in the next cycle to model pipeline behavior. Conversely, calling `decode` before `execute` results in combinational bypass behavior.

From the above observation, we further discover that specifying the global ordering (Figure 3.1(b)) essentially controls the order of calling `enqueue` and `dequeue` of the buffers in a cycle. **Can we specify the ordering inside the buffer directly so that the order between the functions that call `enqueue` and `dequeue` can then be inferred globally?** The answer is positive, and the deductive process with explictly specified local constraints between `enqueue` and `dequeue` methods is shown below. Simply flipping the local ordering constraints allows the designer to model combinational behavior *with the same set of methods* without any other modifications.

$$\left.\begin{array}{c} \texttt{q.dequeue}\ \textbf{precedes}\ \texttt{q.enqueue} \\ A\ \text{calls}\ \texttt{q.dequeue} \\ B\ \text{calls}\ \texttt{q.enqueue} \end{array}\right\} \Longrightarrow \begin{array}{c} A\ \text{precedes}\ B \\ (A < B) \end{array}$$

3.3.3 Achieving Both Fidelity and Modularity

We use the processor/accelerator example in Figure 3.1 to explain how Challenge #1 in Section 3.2 can be fully addressed by explicit ordering constraints. We first create a pipeline queue which specifies { `dequeue` < `enqueue` }. Then we instantiate it between the stages in `Proc` and `Accel`. The global scheduler can automatically deduce the reversed invocation order of Figure 3.1(b–c) without the designer-written tick methods. To accurately model the communication between the processor and the accelerator in Figure 3.1(a), we also need to put two queues inside `Accel` as the communicating buffer for `Accel::work` and `Proc::writeback`, and for `Proc::decode` and `Accel::interface`. For the former pair, since `Accel::work` and `Proc::writeback` are not in the same module, we need to expose the "pointer" of the `dequeue` method from `Accel` to the parent module `Tile` (similar to SystemC `sc_export`) and pass it into `Proc` so that `Proc::writeback`

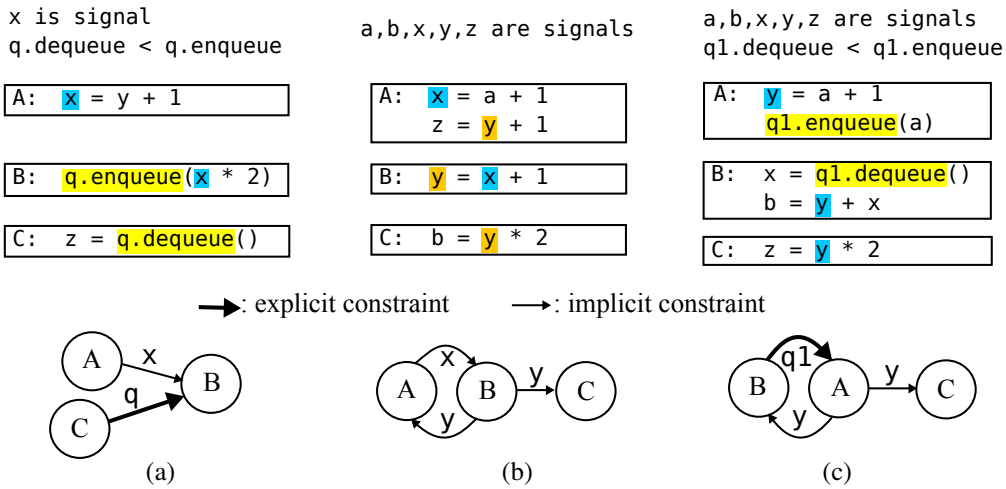


Figure 3.2: CL and RTL Process Examples using UMOC – Code of CL/RTL processes and corresponding unified directed graphs: (a) CL/RTL constraints can co-exist; (b) cycle of RTL processes; (c) cycle of CL processes.

actually calls the `dequeue` method of the queue in `Accel`. The latter pair can be handled similarly by exposing the `enqueue` method from `Accel`.

The global scheduler then automatically deduces { `Proc::writeback` < `Accel::work`, `Accel::interface` < `Proc::decode` }. The designer does not need to write `Tile::tick` and `Top::tick` like Figure 3.1(d–f) at all. A feasible global schedule is able to achieve the same *model fidelity* as flattened tick functions like Figure 3.1(e–f). Moreover, the *modularity* is preserved at the same time. `Accel` module now exposes a `dequeue` method and an `enqueue` method to the outside world, which means we can use the accelerator *as a standalone module* to build other systems without knowing any detail inside `Accel`. Any CL process P that calls the exposed `dequeue` automatically results in an ordering constraint $\{P <$ `Accel::work`$\}$.

3.3.4 Unified Directed Graph (UDG)

The key to solve Challenge #2 in Section 3.2 is to create a unified directed graph (UDG) $G = (V, E)$ where V includes all the hardware processes and E includes all the implicit/explicit ordering constraints between them.

Creating the Unified Directed Graph – For any mixed CL/RTL design, applying the deductive process in Section 3.3.1 and 3.3.2 establishes the preceding relationships not only between all pairs of RTL processes and all pairs of CL processes, but also CL and RTL processes. Figure 3.2(a) shows three hardware processes A, B and C, and the corresponding graph. A writes signal x. B

reads signal x and enqueues a message to the buffer q with pipeline behavior. C dequeues a message from q. We can deduce two ordering constraints in Figure 3.2(a): $\{A < B\}$ from signal x and $\{C < B\}$ from { `q.enqueue` < `q.dequeue` }. Here, B serves as the "glue" between the CL and RTL portions of the design by accessing signals and calling methods at the same time. Note that G may contain cycles. UMOC allows the UDG to have cycles among *only combinational RTL processes* and defers the combinational loop detection to the real simulation if the signal values fail to stabilize. However, UMOC does not allow cycles that include any CL process, because CL processes are usually modeled to execute once per clock cycle due to the side effects on high-level data structures. For example, executing process A of Figure 3.2(c) multiple times may unexpectedly enqueue many elements into $q1$.

Scheduling the Unified Directed Graph for Simulation – The UMOC scheduler schedules the execution of the unified directed graph in each clock cycle. We cannot directly reuse canonical event-driven RTL scheduling algorithms for unified CL/RTL scheduling. This is again because CL processes usually use high-level data structures instead of signal/ports which makes the scheduler hard to trigger subsequent CL/RTL processes, and CL processes are usually modeled to execute exactly once per cycle (see Figure 3.1(g)). Essentially, a correct execution of G must guarantee that before executing any CL process, *all* preceding processes should have been executed, and the cycles of preceding RTL processes have stabilized.

If G contains no cycle, i.e., G is a directed acyclic graph (DAG), a topological sort on G will yield a valid serial schedule. In each clock cycle, we can simply enumerate the serial schedule to execute each hardware process exactly once, satisfying the guarantee for CL processes. Note that there can be multiple possible schedules generated by a topological sort that all result in correct execution [JIB18]. If G contains cycles, according to classic graph theory, a “cycle” in a directed graph is defined as a strongly connected component (SCC) in which every vertex is reachable from every other vertex [Sha81, Tar71]. The scheduler can apply classic SCC algorithms to transform G into a DAG G' of SCCs. Each SCC represents a single vertex in G' or a “cycle” in G. Applying a topological sort on G' yields a serial schedule of all the SCCs. During simulation, we execute all the SCCs in the schedule in each clock cycle. For single-node SCCs, we execute the only hardware process. For multi-node SCCs, we need to iteratively execute all the RTL processes until the signals stabilize and report a combinational loop when it fails to converge.

3.4 UMOC Implementation in PyMTL3

In this section, I present the UMOC implementation in PyMTL3 [JPOB20]. and then discuss how a PyMTL3 hardware description with these primitives can be elaborated to form a unified directed graph and schedule for simulation. Note that the proposed UMOC approach is generic and can be either implemented in any language as a new unified CL/RTL modeling framework, or integrated into existing frameworks to provide the unified CL/RTL modeling capability. Leveraging Python's productive language features, I implement a set of modeling primitives for the designer to construct CL/RTL models, and to capture the signal-based implicit ordering constraints in Section 3.3.1 and method-based explicit ordering constraints in Section 3.3.2 in a modular way. I implement UMOC as the intra-cycle mechanism and cycle-by-cycle simulation as the inter-cycle mechanism. Then I implement PyMTL3 passes to build and schedule the unified directed graph for simulation. I first introduce the proposed primitives to capture RTL and CL constructs, and then discuss the scheduling for the unified directed graph for meaningful simulation. Figure 3.3 shows six code examples.

3.4.1 Modeling Primitives

Here I explain a minimum set of necessary UMOC primitives to simplify the context. Note that the code snippets are showing the *PyMTL3 design* code that uses these primitives, instead of the framework implementation of these primitives. The framework can also include syntactic sugar on top of these primitives to further improve the productivity of designers.

Components – A PyMTL3 component is a hardware module that includes RTL processes and/or CL processes (Figure 3.3(a–d)). It can also instantiate child components to create a hierarchical hardware model (line 6–7 in Figure 3.3(e)).

Signals and Value Ports – Signals and value ports are instantiated as fields of a component (line 3–4, 6, of Figure 3.3(a–b)). PyMTL3 relies on them to infer implicit ordering constraints. Implicit ordering constraints are inferred from accesses to signals and value (input/output) ports. Value ports are exposed to the parent component. Normal signals are internal. Connecting signals and value ports associates all connected signals/ports with the same value and hence propagates the implicit constraint outside the component, which is the key to modularity.

```python
class RegIncrRTL( Component ):
  def construct( s ):
    s.in_ = InPort (32)
    s.out = OutPort(32)

    s.reg = Wire(32)

    @update_ff
    def seq_reg():
      s.reg <<= s.in_

    @update
    def comb_out():
      s.out @= s.reg + 1
```

(a) RTL RegIncr Unit

```python
class WireIncrRTL( Component ):
  def construct( s ):
    s.in_ = InPort (32)
    s.out = OutPort(32)

    s.wire = Wire(32)

    @update
    def comb_wire():
      s.wire @= s.in_

    @update
    def comb_out():
      s.out @= s.wire + 1
```

(b) RTL WireIncr Unit

```python
class RegIncrCL( Component ):
  def construct( s ):
    # Model sequential behavior!
    s.add_constraints(
      M(s.read) < M(s.write),
    )

  @method_port
  def read( s ):
    return s.v + 1

  @method_port
  def write( s, v ):
    s.v = v
```

(c) CL RegIncr Unit

```python
class WireIncrCL( Component ):
  def construct( s ):
    # Model combinational behavior!
    s.add_constraints(
      M(s.write) < M(s.read),
    )

  @method_port
  def read( s ):
    return s.v + 1

  @method_port
  def write( s, v ):
    s.v = v
```

(d) CL WireIncr Unit

```python
class RegIncrCLRTL( Component ):
  def construct( s ):
    s.write = CalleePort()
    s.out   = OutPort(32)

    s.r1 = RegIncrCL()
    s.r2 = RegIncrRTL()

    connect( s.write, s.r1.write )
    connect( s.out,   s.r2.out   )

    @update_once
    def send_to_r2():
      s.r2.in_ @= s.r1.read()
```

(e) CL+RTL Two-Stage RegIncr

```python
class RegIncrRTLCL( Component ):
  def construct( s ):
    s.in_  = InPort(32)
    s.read = CalleePort()

    s.r1 = RegIncrRTL()
    s.r2 = RegIncrCL()

    connect( s.in_,  s.r1.in_  )
    connect( s.read, s.r2.read )

    @update_once
    def send_to_r2():
      s.r2.write( s.r1.out )
```

(f) RTL+CL Two-Stage RegIncr

Figure 3.3: PyMTL3 Buffered Incrementer Units Using UMOC Primitives – (a–b) shows the RTL implementations of a registered incrementer and a wire incrementer using in/out value ports and `update`/`update_ff` blocks. (c–d) shows the CL implementations of a registered incrementer and a wire incrementer using methods and method ports with explicit ordering constraints to specify conbinational/sequential behavior; (e–f) shows the two possible RTL and CL compositions with `update_once` blocks that call method and read/write signals.

Methods and Method Ports – Methods are member functions of a component (line 9–10, 13–14 of Figure 3.3(c–d)). Method ports (including caller and callee ports) are exposed to the parent component. The designer explicitly specifies the ordering constraints that involves methods, which will be collected by PyMTL3 during elaboration. Connecting methods and method ports make all connected method/method ports point to the same method (line 9 of Figure 3.3(e)), allowing the specified constraints to be automatically propagated outside the module.

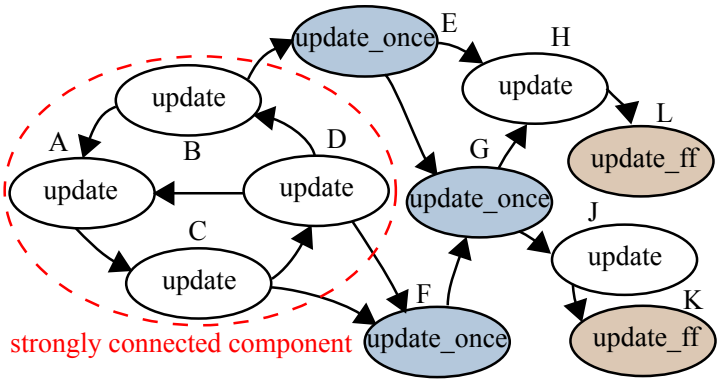
Update Blocks: `update, update_ff, update_once` – PyMTL3 models hardware processes using three types of blocks: `update` for combinational RTL logic (similar to SystemVerilog `always_comb`, `update_ff` for sequential RTL logic (similar to SystemVerilog `always_ff`), and `update_once` for CL modeling. All update blocks can read/write signals and ports, from which the implicit ordering constraints are inferred by PyMTL3. Any signal/port written by a non-blocking assignment in an `update_ff` block is inferred as a sequential element and not counted in ordering constraint deduction. Hence `update_ff` blocks will not precede any other block. In addition to `update` blocks' functionality, `update_once` blocks can also *call methods and method ports*, and hence are restricted to be executed exactly once in each cycle to avoid unwanted duplicate side effects.

Setting Ordering Constraints – Implicit ordering constraints are automatically inferred by PyMTL3. Thus, we do not need to implement any API for setting implicit ordering constraints. I add an API to PyMTL3 for the designer to specify two types of explicit ordering constraints between (1) methods and (2) methods and update blocks. For example, Figure 3.3(c–d) shows the constraints set between two methods: `read` < `write` for sequential behavior, and `write` < `read` for combinational behavior.

3.4.2 Building the Unified Directed Graph

I implement a PyMTL3 UDG generation pass that takes an elaborated PyMTL3 model and generates the corresponding UDG $G = (V, E)$. V includes all the `update`, `update_ff` and `update_once` blocks, and E includes all the implicit and explicit ordering constraints between those blocks. Figure 3.4(a) shows an 11-node UDG example.

Implicit Ordering Constraints – I implement a two-step algorithm to infer implicit ordering constraints. First, I leverage Python's introspection features to obtain the abstract syntax tree of each update block, look for read/write variables, and turn each variable name into an actual object



(a) A unified directed graph example

```
flip_registers()
while not stable:
  A()
  C()
  D()
  B()
  check_threshold()
E() # After SCC
F() # After SCC
G() # After E,F
H() # After E,G
J() # After G
L() # After H
K() # After J
```

(c) 1-cycle execution

```
1:  procedure TICK ( top )
2:      flip_registers( top )
3:      for each SCC c in top.schedule do
4:          if size(c) == 1 then
5:              Execute the only block b in c
6:          else
7:              count = 0
8:              while outputs from c does not stabilize do
9:                  for each block b in c do
10:                     Execute b
11:                 count = count + 1
12:                 if count > threshold then
13:                     error("Found combinational loop!")
```

(b) Generated *tick* function

Figure 3.4: Example of UMOC's Scheduling and Simulation Scheme – (a) the corresponding graph of a design with 11 update blocks, four of which form a strongly connected component; (b) one-cycle execution trace of the *tick* function; (c) the generated *tick* function.

using Python's reflection features. If an object is of signal/port type, we associate the object with the update block. The second step enumerates all the signals collected throughout the hierarchy to perform the deductive process in Section 3.3.1. For each signal x, we add a unidirectional edge $A \rightarrow B$ to the edge set E if block A writes x and block B reads x and A is not an `update_ff` block.

Explicit Ordering Constraints – As Python methods are objects, I apply the same AST-based approach to obtain what methods each `update_once` block invokes. Then, we assemble the invocations with the explicit ordering constraints specified by the designer and perform the deductive process in Section 3.3.2. Specifically, if block A calls method P and block B calls method Q, and the explicit method/method constraint $P < Q$ exists, we add a unidirectional edge $A \rightarrow B$ to the

edge set. Likewise, if block A calls method P and there is an explicit method/update constraint $P < B$ between method P and block B, we add $A \rightarrow B$ to the edge set.

3.4.3 Scheduling the UDG for Simulation

According to Section 3.3.4, `update` blocks may be executed multiple times in a clock cycle until the signals stabilize, as long as no real combinational loop is detected. If an `update_once` block appears in a loop, part of the design is invalid and the interdependency must be removed by the designer. `update_ff` blocks will only be executed exactly once at the end of each clock cycle.

I implement the strongly connected component (SCC) scheduing algorithm in Section 3.3.4 as a PyMTL3 scheduling pass to condense G into a DAG G' of SCCs (e.g., the "cycle" in Figure 3.4(a) will become a single vertex in G'), followed by a topological sort on G' to produce a linear schedule. The pass also checks that any non-trivial SCC doesn't contain `update_once` blocks. Otherwise, the designer must remove the interdependencies.

Then, the tick generation pass takes the schedule and creates a *tick* function that simulates for one clock cycle as shown in Figure 3.4(b). The pass creates a function $flip\_registers$ for *tick* to call at the rising clock edge to double-buffer all sequential elements that appear in the non-blocking assignments of `update_ff` blocks. All the SCCs in the schedule are then executed. The execution of each SCC is either executing one block or repeatedly executing the `update` blocks until the signals stabilize. If the execution does not converge until it reaches the threshold, a combinational loop is detected. Figure 3.4(c) shows *tick*'s execution for one clock cycle.

Note that this scheduling algorithm is compatible with the simulation techniques proposed in the previous work [JIB18] to achieve high simulation performance in pure Python.

3.5 Case Studies

We present two realistic case studies to showcase the effectiveness of UMOC. The designs used are all implemented in PyMTL3. The first case study includes a processor/accelerator composition similar to the motivating example in Figure 3.1, which demonstrates that UMOC can solve the two challenges in Section 3.2. The second case study includes a larger many-core design as

Mechanism	Composition	#Cycles	Deviation	Remarks
Event-driven	RTL Proc + RTL Accel	565	-	baseline
UMOC	RTL Proc + RTL Accel	565	0%	same as baseline
UMOC	CL Proc + CL Accel	541	4%	due to 3-stage
Manual **Proc<Accel**	CL Proc + CL Accel	416	26%	modular sub-tick
Manual **Accel<Proc**	CL Proc + CL Accel	416	26%	modular sub-tick
UMOC	CL Proc + RTL Accel	541	4%	same as CL+CL
UMOC	RTL Proc + CL Accel	565	0%	same as RTL+RTL

Table 3.1: Simulation Cycle Count Results Under Different Scheduling Schemes for CL/RTL Proc/Accel Case Study

evidence for UMOC's ability to handle larger designs with fine-grained CL/RTL compositions for fast design-space exploration during the iterative development process.

3.5.1 Processor/Accelerator Composition

We implement a classic 5-stage pipelined RTL RISC-V processor, and a 3-stage pipelined cycle-level RISC-V processor which contains only three `update_once` blocks to approximately model the RTL processor (`fetch`, `decode+execute+memory`, and `writeback`). We expect a little timing difference across CL and RTL processors, as different number of stages lead to different stalling behaviors due to read-after-write (RAW) hazards. We also implement RTL and CL Fletcher's algorithm checksum accelerators in PyMTL3. The CL accelerator contains two `update_once` blocks to model the request handling and the actual computation using normal Python functions, where the RTL accelerator implements a fairly complex hierarchical design with eight `StepUnit` instances and a finite state machine. For pure-CL composition, we instantiate cycle-level pipeline queues which already include explicit ordering constraints for the `update_once` blocks in the CL processor and CL accelerator to communicate. Thus we do not need to set *any* constraints in the processor and the accelerator. We are also able to expose and connect the queue methods at the top-level.

Table 3.1 shows the simulated cycle count of various compositions running the same microbenchmark. The rolling checksum microbenchmark contains a 25-iteration loop, with each iteration sending 3 loads to memory and 6 requests to the accelerator, resulting in a total of 314 dynamic instructions. For the pure RTL composition, event-driven simulation finishes in 565 cycles, and UMOC has exactly the same simulated cycle count. For the pure CL composition, the global schedule automatically generated by UMOC is able to achieve 4% cycle count difference,

which is expected due to the simplified 3-stage processor pipeline. To model the "manual modular sub-tick" in Figure 3.1(b–d), we manually create two tick functions for CL processes inside the processor and accelerator. For `P<A`, we invoke processor's tick before accelerators's tick, and `A<P` does the opposite. We verify that the tracing output shows unexpected combinational behavior in both cases in contrast to UMOC. As a result, the simulated cycle count has 26% deviation from the pure RTL composition.

For mixed CL/RTL cases, we insert adapters of "glue" blocks at the CL/RTL boundary. PyMTL3 allows us to create adapters for automatically connecting CL/RTL interfaces, which makes the CL/RTL integration effortless. Simulation results show that the CL processor with RTL accelerator has the same cycle count as CL processor with CL accelerator. Also, the RTL processor with CL accelerator has the same cycle count as the pure RTL composition. This confirms that UMOC can provide seamless CL/RTL composition under the same abstraction without losing any model fidelity.

3.5.2 Many-Core/Cache/Network Composition

We implement a many-core system that consists of a parametrizable number of tiles. Each tile contains a parametrizable number of RV32IMAF cores and data caches, sharing one instruction cache, one integer multiply/divide unit (MDU), and one floating point unit (FPU) via on-chip interconnect networks. Throughout the development process, we extensively use fine-grained CL/RTL mixed compositions enabled by UMOC to facilitate design-space exploration, performance evaluation, and the decision on RTL implementation. The CL models are able to capture the desired cycle-level behavior using UMOC explicit constraints and the scheduling pass. UMOC also enables us to seamlessly integrate existing RTL IP blocks that have been fully tested and prototyped in the past, instead of developing additional CL models. Figure 3.5 shows the many-core system with a CL magic memory. Each block is annotated with the availability of CL, RTL, or both CL and RTL models. We use an elf file loader written in Python as part of the test harness to load various RISC-V binaries for parallel programs with a work-stealing runtime to run on the many-core system.

The purpose of implementing the CL multiplier/divider is for quickly studying the performance to decide the type of RTL unit (pipelined or iterative) and the latency/throughput (number of pipeline stages or processed bits per cycle) needed, when shared by multiple processors. After

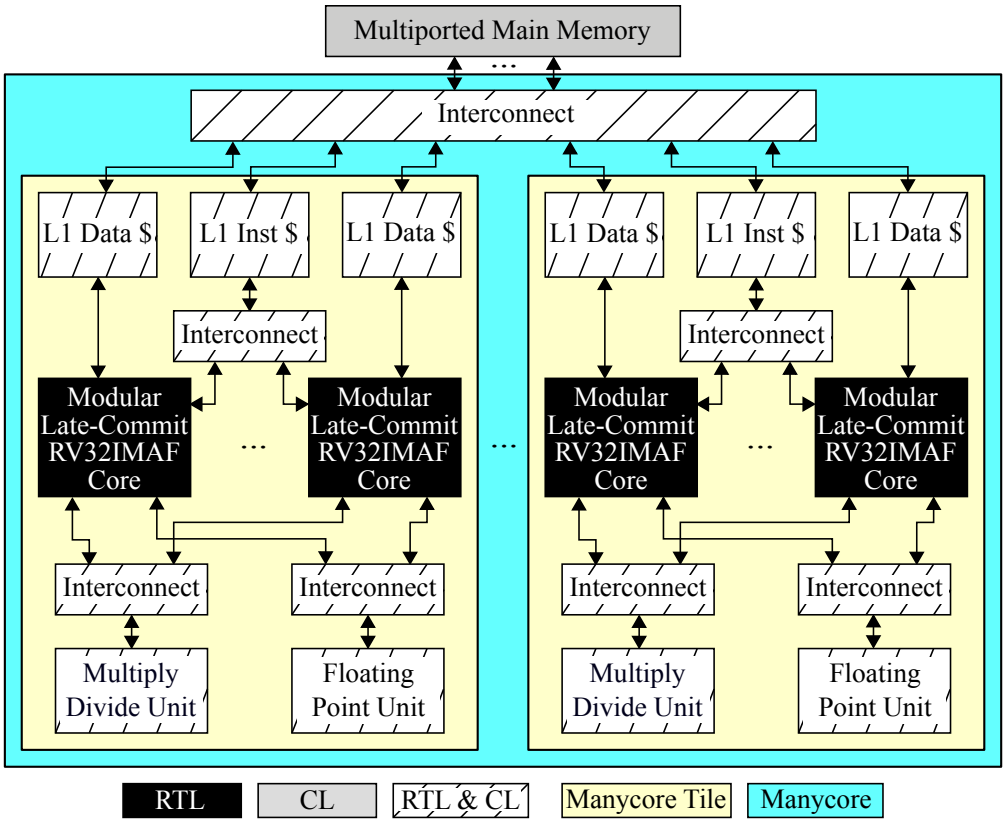


Figure 3.5: Tiled many-core with mixed CL/RTL components – Different colors/patterns show the CL/RTL component availability during the development process. We directly reused the RTL processor, because it was already available prior to the many-core project. We only developed CL model for the main memory, because the main memory is only for testing and verification.

simulating multiple workloads, we decided to implement the iterative divider in RTL because the ratio of div/mod instructions is low. However, we decided to implement a radix-four iterative divider so that each div/mod operation takes 16 instead of 32 cycles, since most division operations are found to stall many subsequent instructions. For the multiplier, we decided to implement a four-stage pipelined multiplier for higher throughput, as some benchmarks contain streams of multiply instructions. The CL models only contain one `update_once` block which processes the request, does the computation, and sends the response to delayed buffers. The user does not need to set any explicit ordering constraints in the multiplier/divider, as appropriate explicit ordering constraints are automatically set when the delay buffers are instantiated with different delays.

As we already developed the RTL processor, developing a CL cache enables quickly exploring the system-level impact of a one-cycle vs. two-cycle hit-latency under different cache sizes and associativities. This influences the parameter selection of different data structures inside the pro-

cessor. The CL cache model only contains several `update_once` blocks which are responsible for composing requests and responses, which is much simplier than the final RTL cache that consists of tens of different components. As Figure 3.5 shows, we have a few different on-chip interconnects in this many-core composition. We are able to develop a single CL network with less than two hundred lines of code to guide the decision of each RTL network implementation. The CL model is essentially a crossbar network, but provides the ability to configure the latency between each pair of input/output terminals, and the size of each terminal buffers, which allows CL model to capture the behavior of more complex network topologies.

3.6 Conclusion

In this chapter, I proposed a novel approach, unified modular ordering constraints (UMOC), to unify cycle-level and register-transfer-level modeling. UMOC addresses the challenges in the state-of-the-art CL modeling approaches and CL/RTL composition approaches. UMOC has been implemented in PyMTL3 as the default modeling mechanism and one of the key features. There have been various hardware IPs and tutorials built using UMOC.

CHAPTER 4
MAMBA++: FRAMEWORK/JIT CO-OPTIMIZATION FOR FAST HARDWARE SIMULATION

The third key challenge in modern hardware modeling frameworks as mentioned in Section 1.2 is *the simulation performance gap in hardware generation and simulation frameworks (HGSF)*. The slow simulation performance in HGSFs' host language undermines the potential benefits of using a productive language for its shorter iterative development cycle in the first place, which makes designers hesitate to adopt these HGSFs and still use HGFs with low-level HDL simulators.

In this chapter, I propose Mamba++, a set of techniques to close the simulation performance gap in Python-based HGSFs. Mamba++ includes JIT-aware HGSF design techniques and HGSF-aware JIT optimization techniques. Using the framework/JIT co-optimization approach, we are able to significantly improve the simulation performance in pure Python and mitigate the Python side of the bottleneck in Python-HDL co-simulation.

4.1 Introduction

The increasing complexity of modern hardware has motivated design teams to augment or even replace traditional domain-specific hardware description languages (HDLs) with high-level general-purpose programming languages. The hope is that high-level languages can reduce time-to-solution by improving the productivity of design and verification. These approaches include: *high-level synthesis* (HLS), where a software-oriented program written in a high-level language is automatically *synthesized* into a low-level HDL implementation [CLN+11, CCA+11, CM08]; and *hardware generation*, where a hardware-oriented declarative or procedural description written in a high-level language is used to explicitly *generate* a low-level HDL implementation. Both approaches use powerful general-purpose language features to improve productivity including: strong static type systems and/or flexible dynamic type systems; object-oriented, generic, and functional programming paradigms; reflection and introspection; lightweight syntax; and rich standard libraries. While both approaches show promise, the focus of Mamba++ is on improving methodologies for highly productive hardware generators. Specifically, Mamba++ improves the simulation performance of hardware models with hardware-based timing semantics.

Early work in hardware generation focused on developing *hardware preprocessing frameworks* (HPFs) which use an ad-hoc intermingling of a high-level language and a low-level HDL (e.g., Scheme mixed with Verilog in Verischemelog [JB99], Perl mixed with Verilog in Genesis2 [SAW+10]). Unfortunately, mixed-language HPFs create a semantic gap, since they require simultaneously designing, verifying, and analyzing designs written in a high-level language (for parameterization, static elaboration, test bench generation) and a low-level HDL (for behavioral modeling). In an HPF, the high-level language usually uses basic string processing and is unaware of hardware semantics. True *hardware generation frameworks* (HGFs) address this semantic gap by completely embedding parameterization, static elaboration, test bench generation, *and* behavioral modeling in a unified high-level "host" language (e.g., Haskell in Lava [BCSS98], standard ML in HML [LL00], Scala in Chisel [BVR+12], Python in Stratus [BDM+07], PHDL [Mas07]). However, HGFs must still generate a low-level HDL implementation for simulation, which prolongs the development cycle and creates a new kind of semantic gap between the high-level host language and the low-level HDL simulation. HDL simulation means designers are limited in the host-language features they can use for online debugging, instrumentation, and profiling. Designers must either manually write test benches in the low-level HDL or use a limited "generator-friendly" subset of the host language to implement test benches. These challenges have inspired completely unified hardware generation and simulation frameworks (HGSFs) where parameterization, static elaboration, test bench generation, behavioral modeling, *and* a simulation engine are all embedded in a general-purpose high-level language (e.g., Java in JHDL [BH98], Haskell in CλaSH [BKK+10], Python in MyHDL [Dec04], PyRTL [CTD+17], Migen [mig], PyHDL [HMLT03]). Our previous work on PyMTL demonstrated the potential for a Python-based HGSF to improve the productivity of hardware design and verification [LZB14].

However, while HGSFs can close the semantic gap present in other approaches, HGSFs also suffer from significantly slower simulation performance. Section 4.2 compares the simulation performance of traditional HDLs, state-of-the-art HGFs, and emerging HGSFs. Our results suggest that for both small and large designs, highly optimized HGSFs are still typically 10× slower than HDL simulation. The highest performing HGSFs use: (1) general-purpose just-in-time (JIT) compilers that are not optimized for HGSFs [BCFR09]; or (2) highly specialized JIT-compiled simulators driven from the host language [LZB14, CTD+17]. Unfortunately, these techniques cannot completely close the performance gap, and/or they reintroduce the semantic gap at an early stage

of the iterative development process (i.e., having the host language for design and another high-level language like C/C++ as the target language for JIT-compilation). This in turn undermines the productivity benefits of using an HGSF.

In this chapter, I propose Mamba++, a set of techniques to close the performance gap in hardware generation and simulation frameworks. *The key insight is the need to deeply co-optimize the HGSF and the underlying general-purpose JIT compiler.* Section 4.3 provides background on tracing just-in-time compilation and state-of-the-art meta-tracing JIT compilers. Then I discuss the Mamba++ techniques in two phases:

1. **Mamba Techniques** – I present several JIT-aware static scheduling and HGSF-aware JIT optimization techniques as the foundation for this work. Section 4.4 describes JIT-aware static scheduling techniques. Section 4.5 describes HGSF-aware JIT optimization techniques. These two sections also quantitatively compares their impact on multiple designs. Section 4.6 compares RISC-V single- and multi-core designs implemented using Verilog, PyMTL [LZB14], and Mamba. Our results suggest static scheduling is able to match the performance of commercial HDL simulators and is 10× faster than existing HGSFs even when simulating more complex designs.

2. **Mamba++ Techniques** – I present the improved Mamba++ JIT-aware scheduling techniques. Mamba++ techniques make the solution to the research question complete and realistic. Section 4.7 reveals the pitfalls of the previous static scheduling techniques in real-world deployment scenarios. Section 4.8 proposes hierarchical static scheduling (HSS). HSS directly reuses the insights from Mamba techniques, and evolves it to address the pitfalls of static scheduling. Section 4.9 evaluates pure PyMTL3 simulation and PyMTL3-Verilator co-simulation using HSS, which demonstrates the practicality of Mamba++ techniques.

Note that the final Mamba++ scheduling algorithm based on strongly connected components [Sha81, Tar71] is actually identical to the UMOC scheduling algorithms. This means Mamba++ scheduling techniques can be directly used on HGSFs that implements UMOC to accelerate mixed CL/RTL simulation. While this work explores these techniques in the context of PyMTL3, our work also sheds light on performance optimization opportunities in other HGSFs.

4.2 Motivation: Simulation Performance Comparison

In this section, I quantitatively evaluate the RTL simulation performance of the four different kinds of hardware development workflows discussed in Section 1.1. The design used in this quantitative evaluation is a 64-bit radix-4 iterative divider implemented at the register-transfer level (RTL) in six different hardware development frameworks (Verilog, Chisel [BVR+12], MyHDL [Dec04], PyMTL [LZB14], PyRTL [CTD+17], Migen [mig]) with different kinds of available simulators (e.g., ahead-of-time compiled, interpreted, JIT compiled). To ensure an apples-to-apples comparison, I implemented the iterative divider in each framework in a very similar way using a structural datapath and finite-state-machine control unit. Figure 4.1 shows the performance of simulating the divider using identical random inputs for 1,000,000,000 cycles assuming the divider is busy: (1) 100% of the time; and (2) only 10% of the time. Source code and evaluation scripts for all designs have been open-sourced at `https://github.com/cornell-brg/mamba-dac2018/`.

Hardware Description Languages – Figure 4.1(a) shows the simulator performance of the hand-written Verilog for the iterative divider. CVS[1], one of the fastest commercial Verilog simulators, achieves 1.2–2.9M simulated cycles/second (CPS). Although CVS does not disclose their internal simulator implementation details, the high raw performance and the big difference between low load and high load implies that CVS uses a highly optimized event-driven simulator. Icarus is an open-source Verilog simulator [ica]. Icarus is well-known to be relatively slow because it first translates Verilog to its internal intermediate language VVP, and then uses an *interpreter* to simulate the design using an event-driven scheme. The resulting CPS is 61K–226K. Note that Icarus is 4× faster under low load than high load, which is also because the interpretation-based scheme is naturally slower than compilation-based scheme. This indirectly confirms that CVS has more optimizations or even a hybrid scheduling scheme for the high-load case. Verilator is an open-source tool for translating synthesizable Verilog into a compiled C++ simulator [ver21]. achieves an impressive 15–18M CPS. This is because Verilator deploys a cycle-based scheduling algorithm which statically schedules all the logic into a single gigantic function and deeply optimize the logic computation within the cycle. Not surprisingly, Verilator does not support simulating designs that leverage pound-delay semantics. Verilator requires C++ testbenches and significantly longer

[1] Tool vendor anonymized due to license agreement.

compile times on larger designs (e.g., several minutes), and hence is more often used for virtual prototyping as opposed to iterative development.

Hardware Preprocessing Frameworks – HPF workflows have similar simulator performance to HDL workflows since they use the exact same HDL simulators. Also note that the HPFs are only responsible for adding parametrization power, and hence the HDL coding style is very similar to the original HDL coding style.

Hardware Generation Frameworks – Figure 4.1(b) shows the simulator performance of the Chisel-generated Verilog for the iterative divider. We can see from the results that HGF-generated Verilog code has similar performance to HDL code. In other words, we can roughly assume HGFs inherit the advantage of using HDL simulators. If we dive deeper into the results, we notice that the performance numbers of HDLs and HGFs are not exactly the same under the same HDL simulator. For example, Chisel-generated Verilog is 20% slower than the handwritten Verilog under Verilator. Under CVS, Chisel-generated Verilog is 20% faster for the low-load case, but 10% faster for the high-load case. This is due to the Scala-Verilog translation process in Chisel forcing a specific coding style in the translation results, which may or may not be favored by the Verilog simulator. Improving the simulation performance of HGF-generated Verilog remains an open research question which requires massive benchmarking or even deeper understanding in simulation mechanism from HGF designers.

Hardware Generation and Simulation Frameworks – The key distinction between HGFs and HGSFs is the ability to use a simulation engine written in the host language to drastically reduce the iterative development cycle and eliminate any semantic gap. The designer avoids crossing any language boundaries for development, testing, and evaluation, and can use the complete expressive power of the host language for verification, debugging, instrumentation, and profiling. Figure 4.1(c) shows the simulator performance of PyMTL for the iterative divider. Simulation using CPython, the reference Python interpreter, is 150× slower than CVS at 100% load. PyMTL uses an event-based simulator that dynamically schedules combinational blocks using an event queue so the average work per cycle is reduced under light load. PyMTL can improve performance by 14–20× using PyPy, a state-of-the-art JIT compiler for general-purpose Python programs [BCFR09]. PyMTL can further improve performance by translating RTL designs into Verilog, translating this Verilog into C++ with Verilator, compiling this C++ into a shared library, and then dynamically linking this library into the original PyMTL program. Overall, PyMTL is

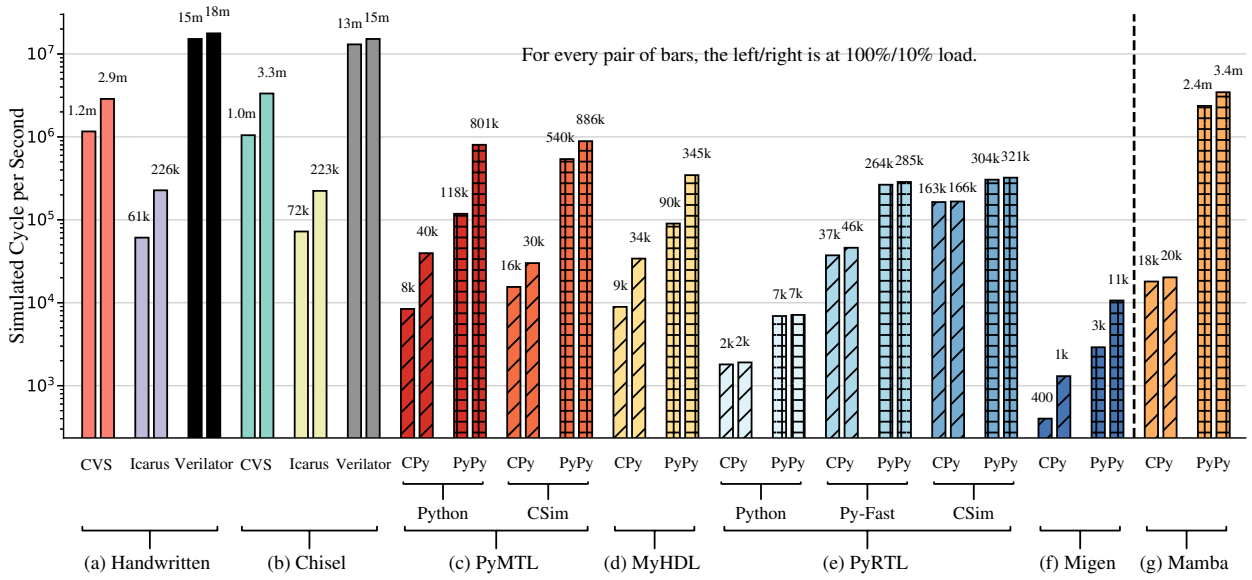


Figure 4.1: Simulation Performance Comparison of Different Hardware Development Workflows – Simulator performance for a 64-bit radix-4 iterative divider implemented at the register-transfer level. Results for identical random inputs for 1B cycles assuming the divider is active: (1) 100% of the time; and (2) only 10% of the time. Chisel = Chisel-generated Verilog; Handwritten = hand-written Verilog; CVS = commercial Verilog simulator; CSim = hybrid C/C++ compiled simulation; CPy = CPython. See Section 4.6 for details on the simulation platform.

able to close the performance gap to less than 10× on this small design, although Section 4.6 suggests slowdowns of ≈10× are more reasonable for larger designs. Figure 4.1(d–f) shows the simulator performance of MyHDL [LZB14], PyRTL [CTD+17], and Migen [mig]. These Python-based HGSFs have their own unique approach to hardware modeling, but all three have dismal performance with CPython and relatively low performance even with PyPy. PyMTL and PyRTL's support for specialized JIT-compiled simulators produces modest performance improvements but also begins to reintroduce the semantic gap by requiring designers to at least on some level interact with multiple languages

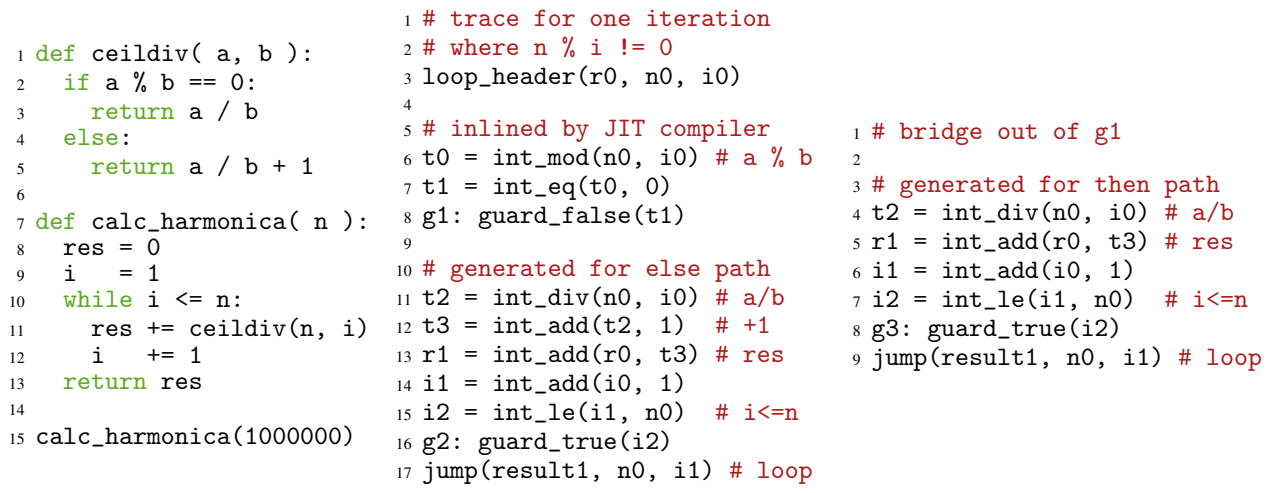
Other Approaches – SystemC [Pan01], a set of C++ classes and macros for system-level design, is also an HGSF, but uses a less productive high-level language compared to Python-based HGSFs. As a result, SystemC is usually used for behavioral simulation and HLS, as opposed to RTL modeling and hardware generation, which is the focus of this work. Bluespec [Nik04] uses a very different approach that combines a new HDL based on guarded atomic actions, limited HLS, and powerful static elaboration mechanisms. This work focuses on less radical approaches to improving the productivity of more traditional RTL design flows.

In summary, Python-based HGSFs suffer from slow simulation performance, which is a major reason why people hesitate to adopt them. In the rest of the chapter, I will discuss how we can close the simulation performance gap between HDL simulators and Python-based HGSFs.

4.3 Background on Meta-Tracing JITs

Many of the high-level programming languages used in HGSFs are *dynamic languages*. Dynamic languages typically include: dynamic typing of variables; lightweight syntax; managed memory and garbage collection; rich standard libraries; interactive execution environments; and advanced introspection and reflection capabilities. These features are critical to the implementation of productive HGSFs, but these features are also the root cause of low HGSF performance. These languages traditionally use interpreters to implement a virtual machine that closely aligns with the language semantics, but as seen in Figure 4.1(c-f), interpreted code can be many orders-of-magnitude slower than statically compiled code. Dynamic languages use JIT-optimizing virtual machines to apply ahead-of-time (AOT) compiler techniques at run-time. Co-optimizing the HGSF and the JIT is the key to achieving peak performance while maintaining HGSF productivity benefits. In this work, we co-optimize the HGSF and the PyPy meta-tracing JIT for Python [BCFR09, AACM07].

Tracing JITs – Tracing JITs start by interpreting the program and profiling the executed code to find frequently executed loops. Upon identifying a *hot loop*, the interpreter records the *trace* of the executed operations of one loop iteration. For better performance through type specialization, the trace also includes the concrete types of variables that were observed as the trace was recorded. This trace is then fed to the optimization engine to generate efficient machine code. Note that the trace is sequential and represents only one of the many possible paths. To ensure correctness, *guards* are placed at every possible point where another code path is possible, e.g., at conditional branches in the executed program or type checks to ensure the actual types match the recorded types. When a guard fails the execution immediately falls back to the interpreter and a new path may be traced and compiled starting from the failing guard if the guard has failed many times. A *bridge* is used to connect the original and new traces. Figure 4.2 shows an example of how tracing JIT works. The code snippet in Figure 4.2(a) executes the `calc_harmonica` function which contains a 1000000-iteration loop. This while loop will be marked as a hot loop after several



```python
def ceildiv( a, b ):
  if a % b == 0:
    return a / b
  else:
    return a / b + 1

def calc_harmonica( n ):
  res = 0
  i   = 1
  while i <= n:
    res += ceildiv(n, i)
    i   += 1
  return res

calc_harmonica(1000000)
```

(a) Python Code

```
# trace for one iteration
# where n % i != 0
loop_header(r0, n0, i0)

# inlined by JIT compiler
t0 = int_mod(n0, i0) # a % b
t1 = int_eq(t0, 0)
g1: guard_false(t1)

# generated for else path
t2 = int_div(n0, i0) # a/b
t3 = int_add(t2, 1)  # +1
r1 = int_add(r0, t3) # res
i1 = int_add(i0, 1)
i2 = int_le(i1, n0)  # i<=n
g2: guard_true(i2)
jump(result1, n0, i1) # loop
```

(b) Initial JIT Trace

```
# bridge out of g1

# generated for then path
t2 = int_div(n0, i0) # a/b
r1 = int_add(r0, t3) # res
i1 = int_add(i0, 1)
i2 = int_le(i1, n0)  # i<=n
g3: guard_true(i2)
jump(result1, n0, i1) # loop
```

(c) Trace of A Bridge

Figure 4.2: Examples of PyPy JIT Trace – (a) Python code of executing one function with a 1000000-iteration loop; (b) JIT trace that PyPy generates for the "else" path as the else path is triggered more frequently during the profiling phase (usually less than a few hundreds of iterations); (c) shows the JIT trace that PyPy generate for the "then" path which follows the guard failure of g1.

iterations, and the JIT will try to optimize the trace for later iterations. Since `n` is not divisible by most `i`, the else path will be triggered more frequently. Figure 4.2(b) shows the trace generated for the else path. The `ceildiv` function call is directly inlined into the loop body, and the loop is optimized to low-level JIT IR code. The divisible check is turned into the guard `g1`. The loop exit check corresponds to the guard g2. Figure 4.2(c) shows the trace generated for the case where `n` is divisible by `i`; note that the loop body does not contain the `+1` statement. This bridge trace essentially branches out of `g1`. Upon any failure of `g1` in later iterations after the bridge trace has been compiled, the JIT compiler will directly jump to the bridge trace without recompilation.

Meta-Tracing JIT – Normal tracing JITs need to be specifically designed for each language in addition to writing the interpreter. This means that even if the programmer wants to add some small feature to the interpreter, he/she needs to learn about the JIT compiler and then modify it to support the added features. PyPy uses a "meta-tracing" JIT approach to build its tracing JIT compiler. Unlike a traditional tracing JIT compiler that records the executed operations in the application, the meta-tracing JIT compiler records the operations performed by the *interpreter as it interprets the application*. This approach separates the complicated JIT compiler machinery from the interpreter implementation and allows easily re-targeting the JIT compiler for other application languages or extensions. In PyPy's case, the interpreter is described in a statically typed subset of the Python language called RPython, and the RPython toolchain will *automatically* attach the meta-tracing JIT

compiler to the interpreter. See [BCFR09, AACM07] for more details on the PyPy meta-tracing JIT. In this work, using a meta-tracing approach drastically simplies the implementation of HGSF-aware JIT optimization techniques. We only need to write RPython code and add hints to the constructs without messing around with the JIT engine.

JIT Warm-Up – Traditional ahead-of-time (AOT) compilation spends time generating optimized machine code in a binary executable file before the real execution. Although a JIT compiler can generate code without AOT compilation, it can spend significant time interpreting, analyzing, and tracing various code paths before actually generating JIT-compiled machine instructions for a frequently executed loop. The actual execution performance during the trace generation phase can be much lower than the steady state, which is often refered to as "JIT warm-up" overheads. JIT warm-up overheads are a key source of overheads at the beginning of execution, especially when there are too many possible hot paths in the code. In this case, the JIT warm-up time can be extremely long due to the exponential explosion of bridges.

Steady-State JIT Execution – The beauty of using a JIT compiler is to hopefully amortize the JIT warm-up overheads by spending most of the steady-state execution time in JIT-compiled code for long-running programs. After the JIT warm-up phase, most traces have been compiled, optimized, and stored in the main memory. The JIT compiler can just invoke the corresponding trace upon any guard failure. Although there may occasionally be some loop iterations that unveil new paths, most iterations can reuse the already compiled traces to maximize the performance. It is worth noting that the difference between the warm-up phase and the steady-state phase requires extra efforts from researchers to appropriately analyze and understand the execution performance.

4.4 Mamba JIT-Aware HGSF Design Techniques

In this section, I present the Mamba JIT-aware HGSF design techniques based on static scheduling. The techniques have been implemented in PyMTL3 as passes. We reuse the PyMTL3 passes to construct sensitivity information based on readers/writers of the same variable in different blocks.

Figure 4.1(g) shows the simulator performance of Mamba for the iterative divider. At 100% load, Mamba is 2× faster than CVS, 20× faster than PyMTL, and 8× faster than PyRTL. The key to Mamba's performance is its co-optimization of the HGSF and JIT which results in a speedup

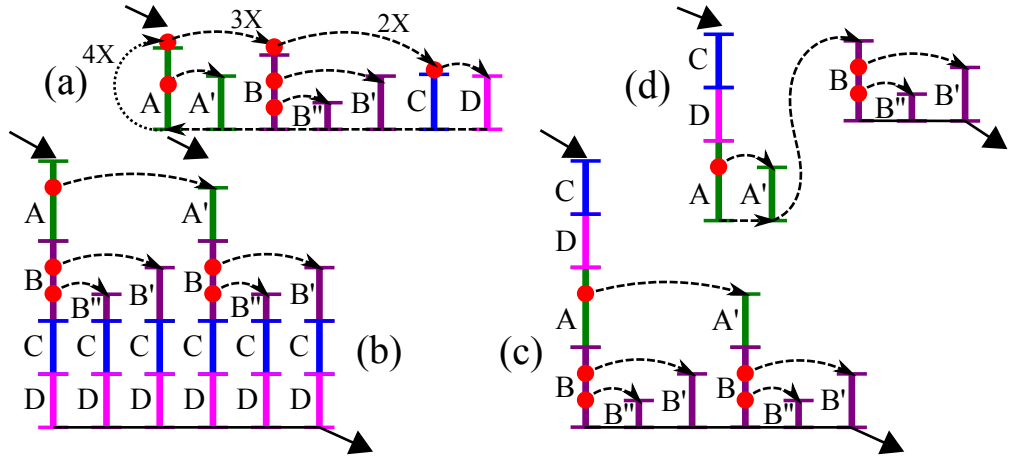


Figure 4.3: Meta-Traces of One Simulated Cycle – (a) event driven and static scheduling; (b) schedule unrolling; (c) heuristic topological sort; (d) trace breaking. A,B,C,D = traces of update blocks; red dots = guards that have bridges compiled from (connected by dashed arrows); A',B',B" = conditional paths in update blocks that result in bridges; 4X,3X,2X = how many times the jump occurs in a simulated cycle; solid arrows = entry from and exit to the cycle loop.

of 124× speedup over CPython. Table 4.1 lists the five JIT-aware HGSF techniques and the two HGSF-aware JIT techniques and reports the incremental performance improvement of each technique. Table 4.1 includes results for the iterative divider from Section 4.2 and a simple single- and multi-core RISC-V design described in more detail in Section 4.6.

As a starting point, we implemented event-driven simulation in Mamba using a very similar technique to PyMTL. Table 4.1 shows the performance of event-driven Mamba simulation for the iterative divider. Like PyMTL, we use two nested loops: an outer loop for simulated cycles, and an inner loop over an event queue of combinational update blocks. Because each iteration of the inner loop is a different update block, the tracing JIT compiles a different trace for each of these update blocks. The tracing JIT will then insert a guard at the beginning of each trace to check if that trace is compiled for the called update block. Figure 4.3(a) illustrates this scenario using a cartoon representation of traces, guards, and bridges. Unfortunately, these guards create a pathological chain of bridges for the inner loop. Executing the n-th compiled update block will result in failing the first $n-1$ guards. In other words, the number of guard failures in an entire simulated cycle scales *quadratically* with the total number of update blocks, which becomes the scaling bottleneck. Small traces for each individual update block also prevents the compiler from performing escape analysis to remove unnecessary memory operations. Finally, enqueuing dependent blocks only when a signal's value changes requires an extra data-dependent check after every assignment. So while event-driven simulation can be efficient when most signals are stable, it can also create a

perfect storm of challenges for tracing JITs. The JIT-aware HGSF techniques described in this section help mitigate many of these challenges.

Static Scheduling – Instead of event-driven simulation, Mamba statically schedules update blocks. While static scheduling has been shown to improve the performance of C++-based simulation frameworks [PMT04, GTBS13], we argue that static scheduling is particularly important in Python-based HGSFs for two reasons: (1) static scheduling avoids bridges due to data-dependent checks on every signal assignment; and (2) static scheduling paves the way for using additional techniques to increase the length of each trace. The Mamba execution semantics require each update block to be executed exactly once in each cycle. This enables a static fixed-order linear schedule to be generated at elaboration time. We leverage the sensitivity information to schedule the update blocks correctly: an update block that writes x should be scheduled before all blocks that read x. We use a topological sort to serialize the dependency graph into a total order of blocks. The topological sorting can succeed only if the directed graph is acyclic (DAG). Thus designers must not create inter-dependencies between combinational blocks. The inner loop simply iterates over the static schedule. Note that this does not change the meta-trace patterns in Figure 4.3(a); this simply changes the way in which execute the corresponding update blocks. Table 4.1 shows that this approach improves the performance by 1.1–13× over event-driven simulation. The concern for static scheduling is that all update blocks are executed regardless of activity. However, a tracing JIT can still optimize a hot path used under light load to improve performance. As shown in Figure 4.1, Mamba is 1.5× faster under 10% load vs. 100% load.

Schedule Unrolling – Static scheduling makes it possible to eliminate the pathological chain-of-bridge pattern in the inner loop by unrolling this loop into a sequence of update block calls. Table 4.1 shows that this improves performance by 1.2–9× compared to static scheduling without inner-loop unrolling for the divider and the 1-core design. Figure 4.3(b) illustrates how static scheduling and schedule unrolling get rid of the chain-of-bridge pattern but increase the overall trace length.

Heuristic Topological Sort – Unfortunately, schedule unrolling can create an *exponential number of bridges* due to data-dependent control flow within each update block. Every code path permutation due to control flow in update blocks (A/A' and B/B'/B" in Figure 4.3(b)) can create a new bridge. In other words, schedule unrolling introduces a new pathological pattern that can lead to serious performance degradation in larger designs (see 32-core in Table 4.1). To address this

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

Table 4.1: Mamba Performance – The baseline is event-driven simulation in Mamba. Each row adds a new technique upon all previous ones. All results are with PyPy. CPS = simulated cycles per second.

problem, we observe that there are multiple valid topological sorts for any given DAG, and each ordering can produce different guard/bridge behavior in our scenario. For example, Figure 4.3(c) illustrates an ordering with fewer guards and bridges (and a smaller instruction-cache footprint) compared to Figure 4.3(b). We use a heuristic to schedule update blocks with potentially more guards as late as possible. The stack used in the topological sort is replaced with a priority queue where each update block's priority is the number of `if`/`elif` statements in that block counted using AST self-parsing. Table 4.1 shows that this can improve the performance by 10–30% over basic schedule unrolling.

Trace Breaking – A large number of guards and bridges is still possible in more complex designs. To further control the number of guards and bridges, we use Python-level JIT hints to break long traces into multiple smaller traces. These application-level hints are provided by PyPy to control the JIT compilation process, and they can be used to prevent tracing in certain parts of the application. During the topological sort, we pack update blocks into a *meta-update block*. A meta-update block is a sequence of one or more update blocks that do not include any `if`/`elif` statements followed by a final update block which does include an `if`/`elif` statement. A meta-update block ends with a trace-breaking hint. This technique essentially limits the number of `if`/`elif` statements within any given trace (see Figure 4.3(d)). Table 4.1 shows this technique has a more significant impact on larger designs, e.g., improving performance by 5× for the 32-core design over heuristic topological sort.

Block Consolidation – Despite the techniques described above, the size of JIT-compiled code scales with the design size due to the nature of JIT compilation: the same update block from

different instances is JIT-compiled individually. This problem is less prominent in static languages because different instances of the same module will likely reuse the same compiled assembly code. Block consolidation is a new technique that deduplicates different instances of an update block in a JIT trace. We modify the topological sort to identify different instances of the same update block and to then schedule these instances together. We group them into a new nested loop that iterates over these different instances by calling the same update block with different parameters in each iteration. Table 4.1 shows that large designs can significantly benefit from block consolidation, e.g., improving performance by 28× for the 32-core design over trace breaking.

4.5 Mamba HGSF-Aware JIT Optimization Techniques

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

Meta-Tracing the Performance-Critical Constructs – Although the PyPy JIT compiler can run arbitrary Python code, native Python constructs may not be the best fit for HGSFs. For example, fixed-bit-width data types are used extensively in HGSFs, but they are not natively supported by Python. HGSF designers must emulate slicing and two's complement arithmetic using integer arithmetic. This increases warm-up time, requires redundant arithmetic operations, and creates excessive bridges due to dynamic type casting. We implement a fixed-bit-width data type in RPython as a proof of concept. Other performance-critical constructs (e.g., byte-addressable memory) can also be implemented in RPython. The key is the meta-tracing approach that enables writing Python-like code exactly once. We exploit the invariant that the bit-width of a signal does not change during simulation; RPython enables annotating the bitwidth as immutable. We are also able to directly manipulate the underlying integer arrays at the RPython level. These specializations significantly eliminate potential bridges. Table 4.1 shows that this technique improves the performance by an additional 1.5–3.5× on top of the JIT-aware HGSF techniques.

Support for Huge Loops – The techniques described in Section 4.4 improve performance but also often increase the total size of all traces. PyPy's VMProf tool is only useful for identifying Python-level bottlenecks, so we use the Linux *perf* tool to identify the microarchitectural impli-

cations of these larger instruction cache footprints. Experiments show that for the 8-core (1-core) simulation in Section 4.6, 3% (0.2%) of all instruction fetches incur an instruction TLB load, among which 22% (2.6%) are iTLB misses. The need for larger TLB reach motivates us to modify PyPy to allocate 2 MB huge pages for traces and to fall back to 4 KB pages if Linux's huge-page support is unavailable. As a proof of concept, the removal of excessive iTLB accesses (confirmed by *perf*) improved the performance of the 32-core design by 10% as shown in Table 4.1.

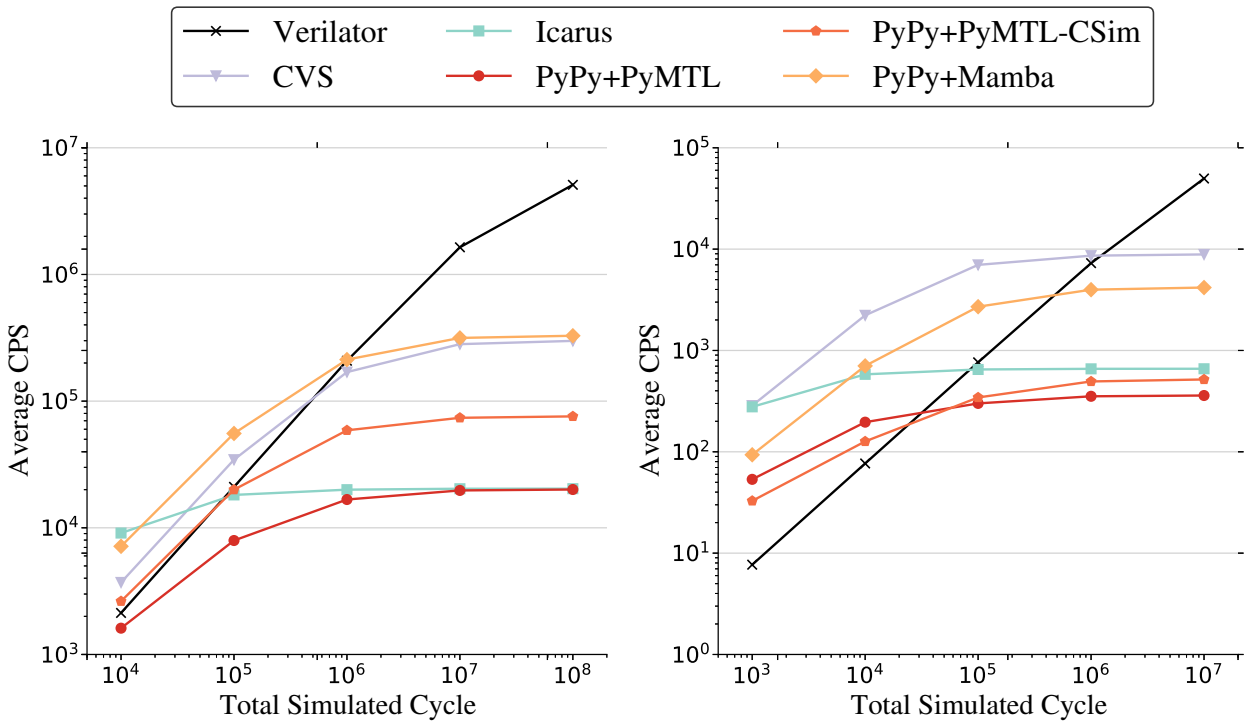
4.6 Case Study for Mamba Techniques

In this section, I present an apples-to-apples simulation performance comparison of 1–32 RTL RV32IM [AP14] five-stage cores implemented in-house in Verilog, PyMTL, and PyMTL3 with Mamba techniques.

4.6.1 Experiment Settings

Design Specification – The RTL RV32IM five-stage cores are implemented using a structural datapath and pipelined control unit in all languages. For static scheduling to be able to schedule the processor, we implement the PyMTL3 processor to be free of cyclic dependencies by dividing some combinational blocks into smaller ones. The cores run a parallel matrix multiplication application kernel using a lightweight parallel runtime. The multi-core does not include caches nor an interconnection network and is simulated with a behavioral test memory implemented in Verilog for CVS and Icarus, C++ for Verilator, and Python for PyMTL and PyMTL3. ASIC synthesis results show that each core can be implemented in around 10 K gates. This design is sufficient for exploring the scalability of various hardware development frameworks, and more complex system-on-chip designs are left as future work.

Simulation Environment – We simulate Verilog with CVS, Icarus, and Verilator, and we use PyPy for PyMTL, PyMTL-CSim, and PyMTL3. The simulation platform includes an Intel E3-1240 v5 processor and 32 GB DDR4-2400 memory running Ubuntu 14.04 Server, gcc-4.8.5, PyPy-5.8, Verilator-3.876, and Icarus-11.0.



(a) Simulating 1-Core

(b) Simulating 32-Core

Figure 4.4: Simulation Performance of RISC-V 1-Core and 32-Core Including Compilation/Warmup Overheads – Each point in (a) and (b) is the average simulated cycle per second (CPS) taking compilation/JIT overhead into account. Basically, the more cycles it simulates, the more the compilation overheads are amortized, and the closer the performance is to the steady state performance.

4.6.2 Results and Analysis

Compilation/Warmup – Figure 4.4(a) and (b) reflect the iterative development cycle for simulating a specific number of instructions. This includes all overheads: CVS, Icarus, Verilator, and PyMTL-CSim compile times; PyMTL and Mamba elaboration times; and PyMTL and Mamba JIT warmup times. Intuitively, the leftmost points (i.e., short simulations) are affected the most by these overheads. Overall, CVS and Icarus have relatively low compilation overhead (1–2 s for 1-core, 3 s for 32-core), whereas Verilator has larger compilation overhead (4–5 s for 1-core, 130 s for 32-core). PyMTL and Mamba have short elaboration times for one core (<1s) but longer elaboration times for 32 cores (6-8s). The JIT warmup overhead is difficult to quantify; both PyMTL and Mamba warm up within at most 10^5 simulated cycles, and the absolute warm-up time is shorter in Mamba compared to PyMTL.

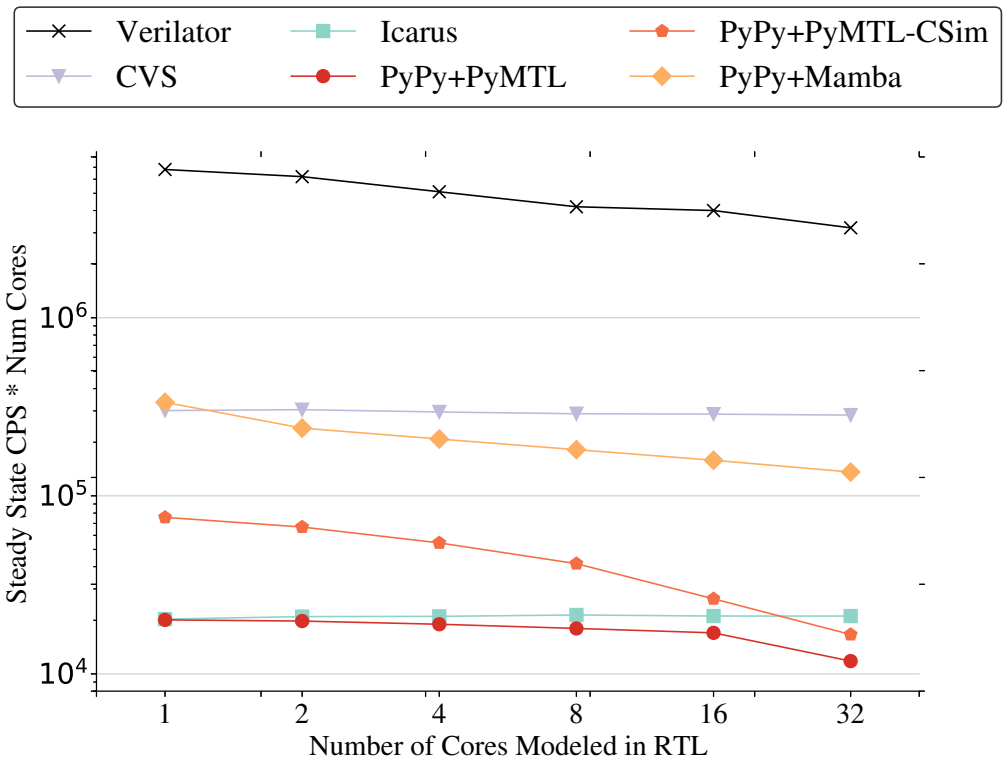


Figure 4.5: Scalable Steady State Simulation Performance of 1–32 RV32IM Cores – Each point in the figure represents the steady state CPS multiplied by number of simulated cores. Since two cores naturally have 2× the complexity of one core, such multiplication will enable us to compare the scalable "system-level" performance across different frameworks.

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

Scalability – Figure 4.5(c) summarizes the steady-state performance of all frameworks with a gradually increasing number of simulated cores. We multiply the simulated cycles per second by the number of cores to reflect the simulation performance scaling with the size of design. A flat line indicates perfect scalability (i.e., a 2× larger design results in a 2× reduction in CPS). CVS and Icarus have good scalability, whereas Verilator appears to be less scalable. The source code

size generated by Verilator scales up linearly with the number of cores, potentially harming the quality of C++ compilation. Mamba is faster than CVS at 1-core, and only 2× slower at 32-core. PyMTL scales better than PyMTL-CSim and Mamba, but its absolute performance is relatively low.

4.7 Pitfalls of Static Scheduling

Previous sections discuss JIT-aware HGSF techniques using static scheduling. This section discusses the pitfalls of static scheduling in realistic scenarios when considered for deployment.

4.7.1 Reduced Modeling Productivity

The previous static scheduling techniques require the graph of all logic blocks to be a directed acyclic graph (DAG). While it is true that real digital circuits will never have any cycles, behavioral logic modeling in HDLs actually allows cycles between two logic blocks as long as there is no actual combinational loop (logic synthesis tools are responsible for transforming directed graphs with cycles to DAG netlists). Figure 4.6(a) shows an example of a 2-bit multiplier where the three combinational blocks follow the dataflow in the circuit to form a simple path. However, Figure 4.6(b) shows the same hardware where two (instead of three) combinational blocks do not follow the natural dataflow and form a cycle. This cycle is valid in terms of HDL semantics, but cannot be handled by static scheduling. This means static scheduling requires the designer to manually rewrite code in order to avoid the cycles, which unfortunately reduces the behavioral modeling productivity.

Figure 4.6(c) shows another example design and implementation of a component that contains a state machine. The whole design implements an enable/ready protocol where any enable signal must factor in the valid signal and AND it with the ready signal. The `comb_state_output` block in Figure 4.6(c) is the most common way for designers to describe a state machine's output signals. All the affected signals are set to specific values for each state in a centralized way, which makes it easier to manage the state machine implementation. Unfortunately, to make the design feasible for static scheduling, the HGSF must require the user to split the state machine output logic block into

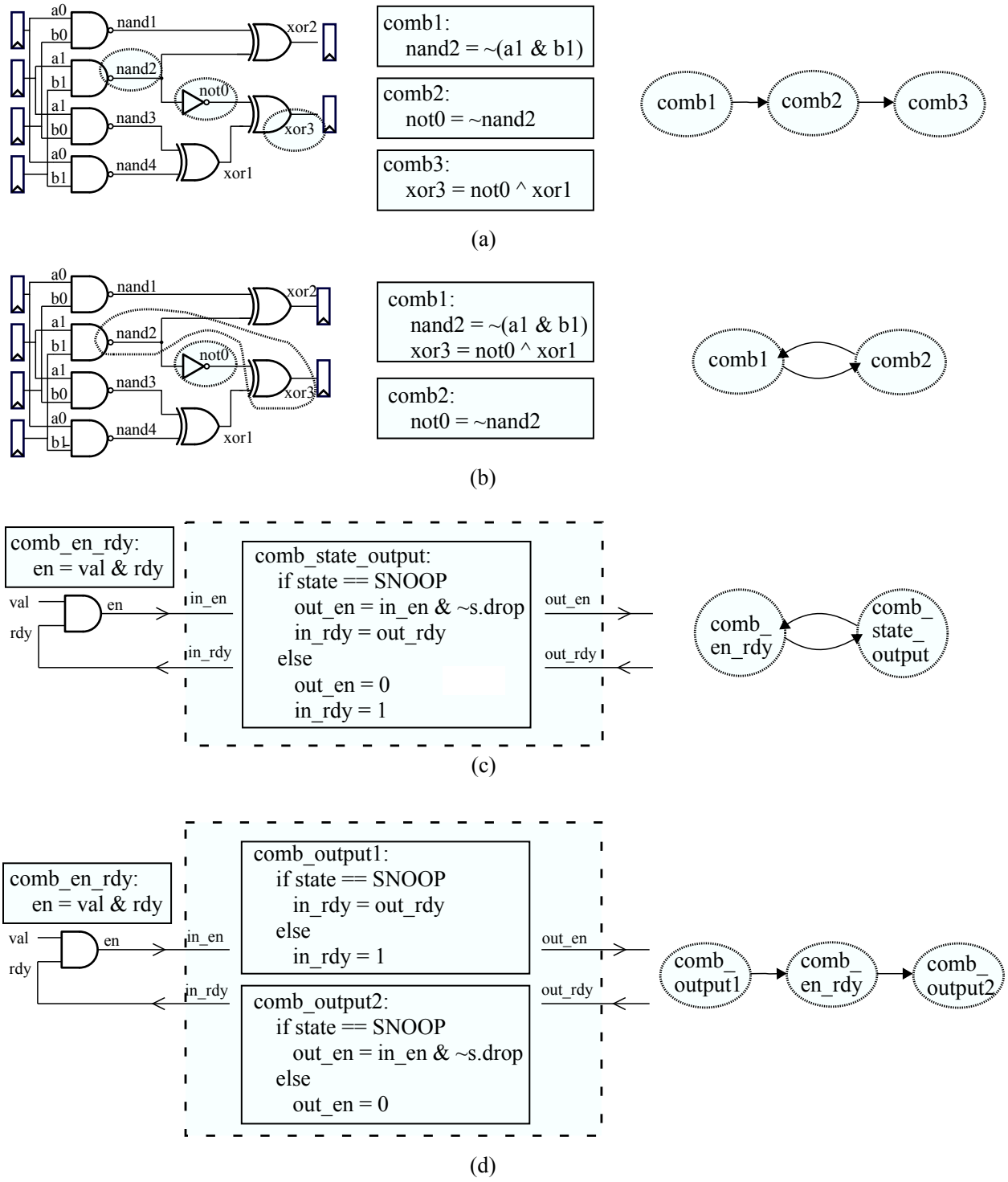


Figure 4.6: Static Scheduling Reduces Modeling Productivity – (a) shows the gate-level netlist of a 2-bit multiplier. Each dotted circle corresponds to one combinational block. The three blocks do not form a cycle. However, (b) shows the same logic modeled by two combinational blocks which form a cycle. Static scheduling cannot handle (b). Static scheduling cannot handle (c) because `in_rdy` is written by `comb_state_output`, and then propagates to be factored into `in_en`, which is read by the same block `comb_state_output`. The designer must rewrite the code like (d) does.

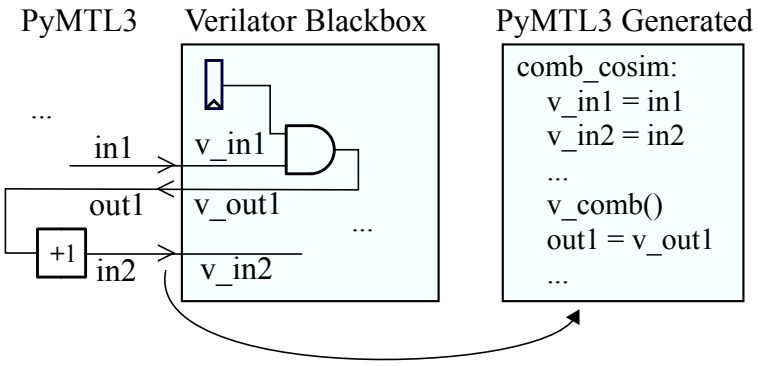


Figure 4.7: An Example of Verilator Blackbox Co-Simulation Which Cannot Be Statically Scheduled – The Verilator-compiled blackbox is part of the simulated composition. `comb_cosim` is the glue block that exchanges the signal values between the PyMTL3 land and the blackbox, and invokes the value evaluation API provided by Verilator. The key issue here is that there is a value dependency going from Verilator land's `v_in1` to `v_out1`, then to PyMTL3 land's `out1` and `in2`, and finally back to Verilator land's `v_in2`. Without knowing any information of the blackbox, the static scheduling algorithm cannot correctly generate and invoke the glue block to correctly exchange values. Specifically, the missing information is: (1) whether there is a cross-boundary value dependency chain; (2) which input/output signals are involved in these chains; and (3) how many times each chain cross the boundary between PyMTL3 and the blackbox.

multiple smaller blocks as shown in Figure 4.6(d). Splitting related logic into two blocks makes the code redundant and confusing, which significantly reduces the behavioral modeling productivity.

I conclude that DAG-based static scheduling restricts the modeling semantics provided by the behavioral modeling language itself, which hurts the designer's productivity in writing RTL code.

4.7.2 Difficulty in Supporting Blackbox HDL Co-Simulation

Because the Python language has very good support for integrating external C/C++ libraries, it is quite common for state-of-the-art Python-based HGSFs [LZB14, myh21, CTD+17] to provide support for Python-C++ co-simulation. For example, PyMTL [LZB14] leverages C foreign function interfaces (CFFI) to import a Verilator-compiled C++ simulator. The main goals of such co-simulation are: (1) to allow the designers who do not want to write RTL in an HGSF or already have developed HDL code to still enjoy the testing/verification productivity brought by an HGSF; and (2) to improve the RTL modeling fidelity of the HGSF by automatically translating and co-simulating valid HDL code. In practice, Python-Verilog co-simulation has played a very important role in research and engineering.

However, DAG-based static scheduling fails to support such blackbox co-simulation even with designers' manual efforts to split all blocks in the Python portion of the model. The fundamental reason is that there can be *combinational* paths going from Python, to the HDL blackbox, and back

```
# e.g., scc_schedule = [ [blk0,blk3,blk4], [blk2], [blk7], [blk1,blk5,blk6] ]
for i in range(100000):
  # generated tick function
  # inlined for illustration

  for scc in scc_schedule:
    if len(scc) == 1: # trivial
      scc[0]()
    else:
      values = record_scc_output_values( scc )

      while True:
        for k in scc:
          k()

        new_values = record_scc_output_values( scc )
        if new_values == values:
          break
        values = new_values
```

Figure 4.8: HSS Baseline Tick Execution – This is a pseudo-code snippet of the baseline HSS tick implementation in Python. Scheduling of the SCCs happen before runtime. The loop body in the code snippet is the scheduled tick function inlined for illustration purposes.

to Python. The combinational path inside the blackbox creates a loop in the full logic graph of both the Python portion of the model and HDL portion of the model, while the Python scheduler cannot assume any property of the blackbox. For example, simply executing the Verilog part of the simulator once each cycle, which is equivalent to assuming all the output signals are combinationally decoupled from the input signals, does not guarantee correct execution. Figure 4.6 shows an example of a Verilog module that cannot be statically scheduled if no information inside the Verilog blackbox is revealed.

I conclude that DAG-based static scheduling algorithms cannot support blackbox HDL co-simulation, which undermines a key benefit of Python-based HGSFs.

4.8 Mamba++: Hierarchical Static Scheduling

To address the pitfalls of static scheduling, I propose *hierarchical static scheduling* (HSS), a JIT-aware scheduling technique to handle arbitrary directed graphs and sustain high simulation performance.

4.8.1 HSS Baseline Algorithm

The foundation of HSS scheduling is the strongly connected components (SCC) algorithm [Sha81, Tar71] to transform an arbitrary directed graph into directed acyclic graphs of SCCs. Each SCC can be either a single logic block or multiple logic blocks that have interdependencies. A topological sort of the SCCs can generate a serial execution schedule. For each cycle of the actual simulation at runtime, we execute the list of SCCs. If the SCC is a single logic block, HSS simply executes it. Otherwise, we need to iteratively execute the blocks in the SCC until all the output variables of the update blocks in the SCC stabilize. Figure 4.8 illustrates the execution. For each cycle, we have a three-level nested loop to: (1) enumerate the SCCs from the topological sort schedule; (2) iterate indefinitely until the output values stabilize; and (3) enumerate the blocks inside the SCC.

Note that the scheduling algorithm coincides with the UMOC scheduling algorithm. As PyMTL3 has implemented UMOC as the preferred modeling mechanism, this means the HSS scheduling algorithm is fully capable of scheduling a hardware model with both CL parts and RTL parts to provide fast simulation performance.

4.8.2 HSS JIT-Aware Optimizations

Compared to simple static scheduling, HSS has two nested loops to execute in one cycle. Note that the nested loops in HSS are inherently different from event-driven simulation. This is because event-driven simulation's while loop pops a different function and executes it in every iteration, but the HSS-generated loop body executes the same set of functions for each iteration. The exit condition of the while loop is based on value change detection.

HSS applies the insights obtained from Mamba JIT-aware HGSF techniques (loop unrolling and trace breaking) to optimize the two nested loops under PyPy. In order to create a simulation tick function that corresponds to a DAC with SCCs, the scheduling algorithm will need to involve similar heuristics to Mamba trace breaking techniques in both levels of loop unrolling to control the number of bridges in each subtrace. For the outer loop, each non-trivial SCC (contains a while loop) can be treated as a non-branchy block, as PyPy will start a new loop trace for the while loop and capture all the branches inside the loop trace. For the inner loop, we use a Hamiltonian path heuristic algorithm to sort the blocks inside the SCC based on the graph topology inside the SCC.

```
# e.g., scc_schedule = [ [blk0,blk3,blk4], [blk2], [blk7], [blk1,blk5,blk6] ]
for i in range(100000):
  # generated optimized tick function
  # inlined for illustration

  values = record_scc_output_values( scc1 )
  while True: # non-trivial SCC
    scc0[0]() # blk0
    scc0[1]() # blk3

    <trace_breaking>

    scc0[2]() # blk4

    new_values = record_scc_output_values( scc )
    if new_values == values:
      break
    values = new_values

  scc1[0]() # trivial SCC, blk2

  scc2[0]() # trivial SCC, blk7

  <trace_breaking>

  while True: # non-trivial SCC
    scc3[0]() # blk 1
    scc3[1]() # blk 5

    <trace_breaking>

    scc3[2]() # blk 6
    ...
  ...
```

Figure 4.9: HSS Optimized Tick Execution – This is a pseudo-code snippet of the tick function with JIT-aware optimization. Compared to baseline, the outermost SCC enumeration loop and the innermost block enumeration loops are unrolled. In each unrolled segments, we insert trace-breaking hints to control the number of bridges in each generated trace.

Then we also apply the Mamba trace breaking techniques to break the inner loop into smaller traces. Figure 4.9 shows the optimized execution.

4.9 Case Study for Hierarchical Static Scheduling

In this section, I present and compare the simulation results for three processor compositions using hierarchical static scheduling. Since the comparison in Section 4.6 among PyMTL3 and other frameworks/HDL simulators has come to the conclusion that PyMTL3 is able to outperform other frameworks and close the gap, I focus on comparing different simulation settings in PyMTL3.

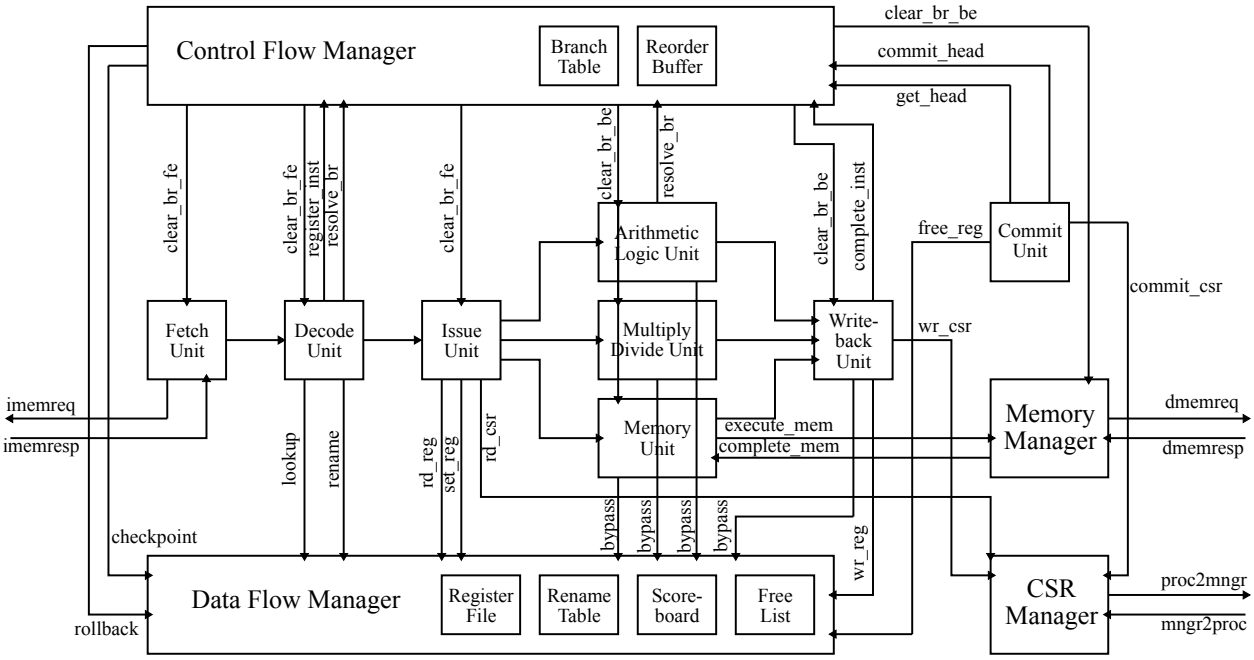


Figure 4.10: PyMTL3 RV32IMAF Modular Processor Diagram – This is the third design I used to evaluate hierarchical static scheduling. Unlike the study in Section 4.6 which duplicates the same core for 1–32 times, this modular processor has many more different components, each of which contains different logic blocks.

4.9.1 Experiment Settings

Design Specification – The three processor compositions are (in increasing order of design complexity): (1) one standalone RTL five-stage pipeline RV32IM [AP14] processor; (2) one RTL five-stage pipeline RV32IM processor with two 2-way associative 8KB RTL blocking caches for the L1 instruction cache and data cache; and (3) one RTL RV32IMAF processor with in-order issue and late commit implemented in a modular fashion and RTL method-based interfaces (block diagram as shown in Figure 4.10). The five-stage processor and the blocking cache are implemented in PyMTL3 using structural datapath and pipelined/FSM control units. The modular processor divides the pipeline into many different sub-units which are composed at the top level. The cores run the same parallel matrix multiplication application kernel as in Section 4.6 using the same lightweight parallel runtime and the same simulator. Note that (1) and (2) are important compositions in ECE 5745 course at Cornell where students simulate these composition throughout all the lab assignments and the final project, and (3) is included in a 14nm chip tapeout. Hence we believe the complexity of the three designs are enough for practical evaluation. The processors/caches are connected to a PyMTL3 cycle-level test memory. which confirms that HSS is fully compatible with the UMOC modeling mechanism.

Design	5-Stage Proc	5-Stage Proc w/ Caches	Modular I2OL Proc
#Vertices	206	556	1440
#Edges	239	704	2275
#Non-Trivial SCCs	2	2	14
Sizes of SCCs	16,5	69,13	51,42,23,21,19,19,19,18,18,17,10,9,8,7

Table 4.2: Unified Directed Graph Characteristics – The number of vertices and edges describes the original UDG. The number of non-trivial SCCs and sizes of SCCs are outcome of the SCC algorithm.

Graph Characteristics – After elaboration, each processor composition turns into an abstract graph of update blocks. According to the UMOC terminology, these graphs are unified directed graphs (UDG). A few important indicators of simulation performance for each design are the number of update blocks in the UDG, the number of non-trivial SCCs in the UDG, and the size of each non-trivial SCC. Note that the impact of the length of each update block can also affect simulation performance, but is hard to quantify. The number and size of non-trivial SCCs are also indicators of how difficult it is to rewrite the design so that it can be scheduled by pure static scheduling. Table 4.2 shows the characteristics of the UDG of each composition. The two 5-stage processor compositions have fewer non-trivial SCCs but each SCC is relatively large. The modular I2OL processor composition has more SCCs, and many of them are relatively large. The analysis of these graph characteristics justifies the need for HSS to handle non-DAG graphs.

Simulation Environment – The simulations are conducted on both CPython and PyPy. I also leverage the Verilog translation/import pass to translate PyMTL3 RTL code into Verilog and use Verilator to compile a C++ simulator similar to PyMTL's CSim mechanism. The hope is that Verilator-compiled C++ simulator can simulate faster in C++. This means each design will be simulated under four settings: CPython, CPython+CSim, PyPy, and PyPy+CSim. The PyPy used here is the same PyPy as Section 4.6 with HGSF-aware JIT techniques.

Moreover, as Verilator's C++ compilation takes quite some time for large designs, I also pick three different GCC optimization flags for the Verilator generated C++ library to explore the trade off between compilation time and simulation performance. The simulation platform includes an Intel Xeon E-2176G processor and 64 GB DDR4-2666 memory running CentOS 7, gcc-4.8.5, PyPy3-7.2, CPython 3.7, and Verilator-4.024.

4.9.2 Results and Analysis

Table 4.3 shows the steady-state simulation performance results of various settings, along with the compilation times of applying different `gcc` options.

CPython vs. PyPy – Not surprisingly, CPython-based simulations are again the slowest among all settings, achieving 205–2330 cycle per second (CPS). PyPy is much faster, achieving 16,500–303,000 CPS for the three designs, which is around 100× faster than CPython. These results are also consistent with the studies we performed in previous sections on static scheduling. Although when going from 5-stage processor to the modular I2OL, CPython has better scalability (11× slowdown compared to PyPy's 18× slowdown), the absolute performance of CPython (205 CPS) is still 80× slower than PyPy's 16,500 CPS. This confirms that pure Python simulation using HSS and HGSF-aware PyPy can bring two orders of magnitudes of speedup for realistic designs in production without any loss of productivity.

Pure Python vs. CSim – Verilator-based CSim is mainly used for verifying that the PyMTL3 RTL code can be correctly translated to Verilog. However, at the cost of slightly losing productivity in debugging, it is an appealing option to accelerate simulation performance as suggested by Table 4.3. CPython+CSim achieves similar performance for all three designs with different sizes, which implies that the C++ part can run much faster than the Python part, and Python part is the bottleneck of execution. This is confirmed by PyPy+CSim which achieves 1.3–6× speedup for the three designs. Note that I2OL results are much slower than 5-stage processor w/ and w/o caches. After profiling the I2OL simulation, I found that the Verilator compiled C++ library for I2OL can achieve around 250,000 CPS, which is only 2.5× of PyPy+CSim (60× of CPython+CSim). This means the HGSF-aware PyPy drastically accelerates the Python part of simulation, making the simulation performance more closed to pure C++ performance. Overall, HSS is able to address the pitfall of blackbox co-simulation and even bring reasonable speedup with Verilator over pure Python simulation.

Under CSim, Why Is Proc With Caches Faster Than Proc Alone? – By horizontally comparing the simulation performance between the 5-stage processor and the obviously more complicated 5-stage processor with two caches, we observe that there is an inversed performance relationship under CSim. Specifically, for pure Python simulation, adding two caches to the processor results in a ≈6× slowdown in CPython (2,330 to 420) and ≈9× slowdown (303K to 33K) in PyPy.

Design	**5-Stage Proc**		**5-Stage Proc w/ Caches**		**Modular I2OL Proc**	
CPS = Cycle Per Second	Simulated CPS	`gcc` Time	Simulated CPS	`gcc` Time	Simulated CPS	`gcc` Time
CPython	2,330	-	420	-	205	-
CPython+CSim **-O3**	3,300	2.85s	6,750	3.9s	4,200	11.72s
PyPy	303,000	-	33,100	-	16,500	-
PyPy+CSim **-O0**	368,000	1.08s	311,200	1.26s	52,300	2.55s
PyPy+CSim **Fine-tuned -O1**	418,900	1.33s	502,300	1.55s	101,800	3.36s
PyPy+CSim **-O3**	425,700	2.88s	531,700	3.92s	102,400	11.80s

Table 4.3: Mamba++ Simulation Results – Each design is evaluated under six different settings. The settings with `gcc` time are Verilator co-simulations. Higher simulated cycle per second is better. Lower `gcc` time is better.

However, it becomes a $\approx 2\times$ speedup (3,300 to 6,750) under CPython+CSim and $\approx 1.3\times$ speedup under PyPy+CSim (419K to 502K and 426K to 532K).

After carefully profiling the execution, we confirmed that this speedup is valid. The fundamental reason is that the two caches are part of the RTL code and get translated into C++, which significantly reduces the number of memory requests handled in the PyMTL3 cycle-level memory. If only the 5-stage processor is connected to the CL memory, the processor will send one instruction fetch request per cycle and one data fetch request every few cycles to the CL memory. However, when combined with two caches, these fetch requests are handled by the cache first, which results in significantly fewer requests sent out to the CL memory (only cache misses are sent out). As we already understand from previous discussions, the Python side of execution is actually the bottleneck compared to the Verilator generated C++ which is overwhelmingly faster than the Python part. In summary, the performance improvement due to the reduction of activity in PyMTL3 CL memory significantly overweighs the slowdown in the C++ simulation due to the increase in RTL design size.

This intricate performance issue can possibly open up more research opportunities in extracting more simulation speedup by offloading appropriate computations to the C++ part.

Impact of `gcc` Optimization Options – Verilator-based CSim brings a trade-off between compilation time and compiled execution performance. A higher optimization level of `gcc` can bring better performance, but requires longer time to compile. In order to determine the optimal set of options, I manually disable specific optimizations on top of the generic `-O0`, `-O1`, `-O2`, `-O3`, `-Os` optimization flags (also note that some optimizations cannot be turned off if `-O1`/`-O2`/`-O3` are applied). To compare the effect of different optimization levels, I careful pick three set of options to compare the performance: `-O0`, i.e. turning off all additional optimizations; fine-tuned

```
-O1 -fno-guess-branch-probability -fno-reorder-blocks -fno-if-conversion
-fno-if-conversion2 -fno-dce -fno-delayed-branch -fno-dse -fno-auto-inc-dec
-fno-branch-count-reg -fno-combine-stack-adjustments -fno-cprop-registers
-fno-forward-propagate -fno-inline-functions-called-once -fno-ipa-profile
-fno-ipa-pure-const -fno-ipa-reference -fno-move-loop-invariants
-fno-omit-frame-pointer -fno-split-wide-types -fno-tree-bit-ccp
-fno-tree-ccp -fno-tree-ch -fno-tree-coalesce-vars -fno-tree-copy-prop
-fno-tree-dce -fno-tree-dominator-opts -fno-tree-dse -fno-tree-fre
-fno-tree-phiprop -fno-tree-pta -fno-tree-scev-cprop -fno-tree-sink
-fno-tree-slsr -fno-tree-sra -fno-tree-ter -fno-tree-reassoc
```

Figure 4.11: Fine-Tuned gcc Optimization Options Based on -O1 – The listed gcc compiler options are mostly removing specific optimizations from the default -O1 set. The goal is to reduce compilation time without slowing down the simulation performance.

`-O1`, i.e. `-O1` with some options turned off as listed in Figure 4.11; and (3) `-O3`. Table 4.3 shows the simulation performance and compilation time. We can conclude that: (1) `-O3` provides the fastest simulation performance and longest compilation time, but the compilation time drastically increases when the design becomes bigger; (2) `-O0` results in very low compilation time and reasonable performance; and (3) our customized `-O1` option takes ≈20% longer than `-O1` to compile for all three designs, but is able to achieve the `-O3` level of simulation performance. These insights lead to our decision to deploy the custom `-O1` option in production for longer simulations, and `-O0` option for short simulations.

4.10 Conclusion

This chapter presents Mamba++, a set of techniques to close the simulation performance gap in Python-based hardware generation and simulation frameworks. The key insight of this chapter is the need to deeply co-optimize the HGSF and the underlying general-purpose JIT compiler. Static-scheduling-based Mamba techniques including several novel JIT-aware HGSF as well as HGSF-aware JIT techniques match the performance of a commercial HDL simulator and improve performance compared to prior HGSFs by 10×. Then, Mamba++ addresses realistic deployment concerns with minimum performance loss to accommodate more flexible HDL semantics. I prototyped Mamba/Mamba++ in PyMTL3 by implementing all the scheduling algorithms as PyMTL3 passes and customizing a PyPy JIT compiler. The modified PyPy has been open-sourced at `https://github.com/pymtl/pypy-pymtl3`, and the Mamba/Mamba++ simulation passes have

been open-sourced at `https://github.com/pymtl/pymtl3/tree/master/pymtl3/passes/mamba`. While this paper explores these techniques within the context of PyMTL3, our work also sheds light on performance optimization opportunities in other HGSFs. We hope to break the long-lasting obstacle in HGSF simulation performance that prevents researchers/engineers from adopting HGSFs.

CHAPTER 5
PYH2: PRODUCTIVE TESTING METHODOLOGIES FOR AGILE HARDWARE DESIGN

This fourth challenge in modern hardware modeling frameworks as mentioned in Section 1.2 is reducing testing/verification time for agile hardware design flows. Most academic groups and open-source hardware teams use an agile approach to develop and verify hardware design blocks due to the high cost of hiring dedicated verification engineers. It is crucial to have rapid and comprehensive verification methodologies that reduce testing/verification overheads. Moreover, we believe the success of the emerging open-source hardware/EDA ecosystem critically depends on thoroughly tested open-source hardware blocks.

In this chapter, I present PyH2, our vision for novel productive testing methodologies using open-source hardware generation and simulation frameworks and key open-source software packages in the ecosystem. PyH2 combines the advantages of Python, PyMTL3, and hypothesis to create productive and customized testing methodologies for different categories of hardware designs. Specifically, PyH2 attempts to reduce the designers' effort in creating high-quality property-based random tests. I co-led the work with Yanghui Ou, where Yanghui is responsible for initial efforts on combining PyMTL3 with hypothesis, and exploration of the PyH2 methodology with a focus on PyH2G.

5.1 Introduction

As Dennard scaling is over and Moore's law continues to slow down, modern system-on-chip (SoC) architectures have been moving towards heterogeneous compositions of general-purpose and specialized computing fabrics. This heterogeneity complicates the already challenging task of SoC design and verification. Building an open-source hardware community to amortize the non-recurring engineering effort of developing highly parametrized and thoroughly verified hardware blocks is a promising solution to the heterogeneity challenge. However, the widespread adoption of open-source hardware has been obstructed by the scarcity of such high quality blocks. We argue that *a key missing piece in the open-source hardware ecosystem is comprehensive, productive, and open-source verification methodologies* that reduce the effort required to create thoroughly tested

hardware blocks. Compared to closed-source hardware, verification of open-source hardware faces several significant challenges:

1. Closed-source hardware is usually owned and maintained by companies with dedicated verification teams. These verification engineers usually have many years of experience in constraint-based random testing using a universal verification methodology (UVM) with commercial SystemVerilog simulators. However, open-source hardware teams usually follow an agile test-driven design approach stemming from the open-source software community, where the designer is also responsible for creating the corresponding tests. Moreover, the steep learning curve, in conjunction with very limited support in existing open-source tools, makes the UVM-based approach rarely used by open-source hardware teams. *We argue that the open-source hardware community is in critical need of an alternative route for testing open-source hardware, instead of simply duplicating closed-source hardware testing frameworks.*

2. Unlike closed-source hardware's development cycle where most engineers focus on a specific design instance for the next generation product, open-source hardware blocks usually exist in the form of design *generators* to maximize reuse across the community [SWD+12]. However, design generators are significantly more difficult to verify than design instances due to the combinatorial complexity in the multi-dimensional generator parameter space. *There is a critical need to create an open-source framework that systematically and productively tests design generators and automatically simplifies both failing test cases and failing design instances to facilitate debugging.*

3. Performing random testing can be difficult in important hardware domains. There has been a major surge in open-source RISC-V processor implementations. However, due to limited human resources, most of these implementations only include a few directed tests, randomly generated short assembly sequences, and/or very large scale system-level tests (e.g., booting Linux). *There is a critical need to create an automated random testing framework to improve the fidelity of open-source processor implementations.*

4. Many open-source hardware blocks are designed to improve reusability by exposing well-encapsulated timing-insensitive handshake interfaces that can provide an object-oriented view of the hardware block (e.g., a hardware reorder buffer exposes three object-oriented

"method" interfaces: allocate, update, and remove). However, it is very hard to perform random testing to test the behavior of concurrent hardware data structures that have multiple interfaces accepting "transactions" in the same cycle. Converting a random transaction sequence into cycle-by-cycle test vectors using traditional testing approaches requires a cycle-accurate golden model. Manually creating multi-transaction test-vectors only works for directed testing. One possible solution is to execute only one random transaction in each cycle, yet the inability to stress intra-cycle concurrent behavior harms the quality of the tests. *There is critical need to create a novel testing approach for object-oriented hardware using concurrent intra-cycle transactions.*

To address these challenges, we introduce PyH2[1], our vision for a productive and open-source testing methodology for open-source hardware, which is significantly different from state-of-the-art closed-source hardware testing. Leveraging open-source software, PyH2 attempts to solve the open-source hardware testing challenge by holistically using property-based random testing (PBT) in Python to significantly reduce designer effort in creating high-quality tests. The advantage of PBT over constraint-based random testing is: (1) PBT does not draw all of the random data beforehand, making it possible to leverage runtime information to guide the random data generation; and (2) PBT can automatically shrink the failing test case to a minimal failing case once a bug is discovered. Compared to BlueCheck [NM15], a prior PBT framework for hardware, the key distinctions are: (1) PyH2 enables using a high-level behavioral specification written in Python as the reference model instead of requiring the reference model to be synthesizable; (2) the random byte-stream internal representation of `hypothesis` provides more sophisticated auto-shrinking, while BlueCheck simply removes transactions along with ad-hoc iterative deepening; and (3) PyH2 can auto-shrink not only the transactions but also the design itself by unifying the design parameter space and the test-case space. We see coverage-guided mutational fuzzing (e.g., RFUZZ [LKK+18]) as complementary to PBT. PBT can be used to quickly find bugs with moderate complexity, while RFUZZ can be used to very slowly find potentially more complex bugs. Overall, PyH2 is able to combine the advantages of complete-random testing and iterative-deepened testing to identify a failing test case quickly and then provide a minimal failing case to facilitate debugging.

PyH2 is supported by the whole Python ecosystem, among which three main packages form the foundation of PyH2 (PyMTL3, `pytest`, and `hypothesis`). PyH2 users can use over 100,000 open-

[1] Python's Hypothesis for Hardware

source Python libraries to build test benches and golden models. PyH2 leverages PyMTL3 [JIB18, JPOB20] to build Python test benches to drive RTL simulations with PyMTL3 models and/or external SystemVerilog models leveraging PyMTL3's Verilator co-simulation support. PyH2 adopts `pytest`, a mature full-featured Python testing tool, to collect, organize, parametrize, instantiate, and refactor test cases for testing open-source hardware. PyH2 also exploits `pytest` plugins to evaluate hardware-specific testing metrics. For example, PyH2 tracks the line coverage of behavioral logic blocks of PyMTL3 models during simulation using `coverage.py`, a line coverage tool for normal Python code. The key component of PyH2 is `hypothesis`, a PBT framework to test Python programs by intelligently generating random test cases and rapidly auto-shrinking failing test cases.

PyH2 is realized by a collection of PyH2 frameworks which are discussed in depth in the rest of the chapter: PyH2G (PyH2 for RTL design generators), PyH2P (PyH2 for processors), and PyH2O (PyH2 for object-oriented hardware).

5.2 Background

This section briefly introduces PyMTL3, `pytest`, and `hypothesis`, the three key Python libraries that form the foundation of PyH2.

5.2.1 PyMTL3

PyMTL3 is an open-source Python-based hardware modeling, generation, simulation, and verification framework. PyMTL3 supports multi-level modeling for register-transfer-level (RTL), cycle-level, and functional-level models. To provide productive, flexible, and extensible workflows, PyMTL3 is designed to be strictly modular. Specifically, PyMTL3 separates the PyMTL3 embedded domain-specific language that constructs PyMTL3 models, the PyMTL3 native in-memory intermediate representation (NIMIR) that systematically stores hardware models and exposes APIs to query/mutate the elaborated model, and PyMTL3 passes that are well-organized programs to analyze, instrument, and transform the PyMTL3 NIMIR.

PyMTL3 aims at creating an evolving ecosystem with its modern software architecture and high interoperability with other open-source tools. PyMTL3 emphasizes performing simulation in the

```python
def gcd( a, b ):
  while b > 0: # bug: while b > 10
    a, b = b, a % b
  return a

# Create two tricky directed test cases
@pytest.mark.parametrize(
   "a,  b,  ref", [
  [ 12, 18, 6 ],
  [ 65, 33, 1 ]
)
def test_directed( a, b, ref ):
  assert gcd( a, b ) == ref
```

(a) Parametrizing directed tests using a pytest decorator

Desired Property	CRT	IDT	PBT
Small number of test cases to find bug	✓	X	✓
Small number of transactions in bug trace	X	✓	✓
Simple transactions in bug trace	X	✓	✓

(b) Comparison of different testing techniques

```python
import math, random, hypothesis

def test_complete_random():
  for _ in range( 100 ):
    a = random.randint(1, 128)
    b = random.randint(1, 128)
    assert gcd( a, b ) == math.gcd( a, b )

def test_iterative_deepened():
  for a in range( 1, 128 ):
    for b in range( 1, 128 ):
      assert gcd( a, b ) == math.gcd( a, b )

@hypothesis.given(
  a = hypothesis.strategies.integers(1,128),
  b = hypothesis.strategies.integers(1,128),
)
def test_property_based( a, b ):
  assert gcd( a, b ) == math.gcd( a, b )
```

(c) Code for testing a greatest common divisor function using complete-random testing (CRT), iterative-deepened testing (IDT), and property-based testing (PBT)

Figure 5.1: Background on Testing Methodologies

Python runtime and automatic `Verilator` black-box import for co-simulation. Driving the simulation from Python test benches to test both PyMTL3 designs and external SystemVerilog modules enables PyMTL3 to combine the familiarity of Verilog/SystemVerilog with the productivity features of Python. Tools that take the opposite approach (e.g., cocotb) embed Python in a Verilog simulator and drive the simulation from the Verilog runtime, but this complicates the ability to leverage the full power of Python. RTL designs built in PyMTL3 can be translated to SystemVerilog accepted by commercial EDA tools, or Yosys-compatible Verilog accepted by OpenROAD, a state-of-the-art open-source RTL-to-GDS flow [ACF+19].

5.2.2 PyTest

`pytest` is a mature full-featured tool for testing Python programs. Using `pytest`, the programmer can create small tests with little effort and also parametrize numerous complex tests with compositions of `pytest` decorators succinctly as shown in Figure 5.1(a). `pytest` also provides lightweight command line options to print out different kinds of error messages varying from a list of characters indicating whether each test fails, to per-test full stack traces. `pytest` has hundreds of plugins, such as `pytest-cov` that leverages `coverage.py` to track line coverage.

5.2.3 CRT, IDT, and Hypothesis PBT

Traditional testing methodologies usually use a mix of complete-random testing (CRT) and iterative-deepened testing (IDT). As shown in Figure 5.1(b), CRT can detect errors quickly because it randomly samples the input space, but can produce very complicated failing test cases which are difficult to debug. IDT finds bugs more slowly because it gradually samples the input space, but can produce simple counterexamples. Property-based testing (PBT), first popularized by QuickCheck [CH00], is a high-level, black-box testing technique where one only defines *properties* of the program under test and uses *search strategies* to create randomized inputs. The original QuickCheck paper also discussed the integration with Lava [BCSS98] to test circuits. Properties are essentially partial specifications of the program under test and are more compact and easier to write and understand than full system specifications. Users can make full use of the host language when writing properties and thus can accurately describe the intended behavior. Most PBT tools support *shrinking*, a mechanism to simplify failing test cases into a minimal reproducible counterexample. With these features, PBT can achieve the benefits of both CRT and IDT.

`hypothesis` [MHDmoc19] is a state-of-the-art Python PBT library that includes built-in search strategies for different data types and supports integrated auto-shrinking of failing test cases. All `hypothesis` strategies are built on top of a unified random byte-stream representation, and each strategy internally repurposes random bytes to produce the target random value. Search strategies in `hypothesis` are integrated with methods that describe how to simplify certain types of data, which makes shrinking effective. Users can compose built-in search strategies for any user-defined data type and shrinking will work out-of-the-box.

Complicated stateful systems can also be tested with `RuleBasedStateMachine` in `hypothesis`. The user inherits from the `RuleBasedStateMachine` class to add variables, a prologue, and an epilogue to create a new test class. The user needs to define `rules` and their `preconditions` and `invariants`, which describes conditional state transitions. For stateful testing, usually the user creates Python assertions inside the rule to compare against a golden reference model. `hypothesis` repeatedly instantatiates the test class and executes a sequence of rules on the state machine.

Figure 5.1(c) shows examples of testing the greatest common divisor function using CRT, IDT, and `hypothesis` PBT against `math.gcd`. The CRT test (lines 16–20) includes 100 random samples. The IDT test (lines 22–25) iteratively tries all possible values for a and b from 1–128. We use the `@hypothesis.given` decorator to transform a normal function `test_property_based` that

accepts arguments, into a randomized PBT test. Consider a bug where line 3 in Figure 5.1(a) is changed to `while b>10`. CRT can find the bug quickly, but the failing test case involves relatively large numbers. IDT finds the bug in exactly 11 test cases (i.e., `gcd(1,11)`). PBT can find the bug quickly with large numbers, but then auto-shrink the inputs to a minimal counterexample (i.e., `gcd(2,1)`).

Fundamentally, the auto-shrinking feature of PBT converts the problem of "finding a minimal failing case" into an optimization problem where the input space is a list of bytes that are interpreted as different test inputs and the optimization goal is to find the minimum amount of bytes that still triggers the bug [MD20]. Internally, PBT's shrinking process leverages algorithms/heuristics like hill climbing, simulated annealing, and gradient descent to remove part of the bytes and then re-run the test.

5.3 PyH2G: PyH2 for RTL Design Generators

PyH2G is a PyH2 framework to productively and effectively test *RTL design generators*. We envision that future open-source system-on-chip designs are heavily based on chip generators which are composed of numerous highly parametrized RTL design generators.

5.3.1 Challenge in Testing RTL Design Generators

Unfortunately, verifying design generators is significantly more challenging than verifying design instances due to the *combinatorial explosion* in the multi-dimensional generator parameter space. For example, to support generating cache instances of different sizes, an RTL cache generator can be parameterized over the word size (e.g., 4–16 bytes), the size of each cacheline (e.g., 2–32 words) and the number of cachelines (e.g., 16–1024 cachelines). Assuming parameter values must be a power of two (which is not always true), the above example results in a whopping number of 105 different design instances. Then, the cache generator may also parameterize over the request queue size (1–8), set associativity (1–16), and miss-status-handling-register (MSHR) size (4–16). Behavior-wise, it may also parametrize over replacement policy, and blocking/non-blocking behavior. The number of possible instances can quickly grow to over a million.

Since the goal of creating RTL design generators is to reduce the NRE cost by reusing the generator to generate different instances, *the whole generator parameter space* must be verified with enough coverage to provide compelling evidence for correctness. Traditional testing techniques such as CRT and IDT face new challenges in covering such large generator parameter space. CRT can find a bug quickly with a few test cases but has no guarantee on the size of the failing case and the failing instance. As a result, CRT often leads to a complicated failing test case with numerous transactions *and a complex design instance*, which makes it more difficult to debug. IDT can produce a simple failing case with a small design instance, but may take a very long time to detect the error due to the iterative deepening required for the generator parameters.

5.3.2 PyH2G Implementation

In response to these challenges, PyH2G smartly leverage property-based testing (PBT) to obtain the benefits of both CRT and IDT. The key idea of PyH2G is to unify the generator parameter space and the test case space during `hypothesis` random data generation. Using the composite strategy interface provided by `hypothesis`, we specify a composite search strategy that includes both the design parameter strategies and the test case generation strategies. Lines 7–10 of Figure 5.2 shows an example of how we apply `@given` decorator on the test function to create composite strategies for one design parameter (`num_terminals`) and one list of test case, and use them as parameters to the test function. Inside the test function, we directly use these parameters to elaborate the test harness as if there is no hypothesis strategy involved.

During runtime, `hypothesis` internally interpret part of the generated random byte stream as the design parameters and the rest as the test case. Such composite strategy and unified random data generation also allows `hypothesis` to simultaneously shrink the design parameters (i.e., reducing the complexity of the generated design instance), the length of the input transaction sequence, and the complexity of each transaction to a minimal failing test case.

5.3.3 Case Study: On-Chip Network Generator

We quantitatively evaluated CRT, IDT, and PyH2G using the PyOCN [TOJ+19] ring network generator against four real-world bugs as described in the table of Figure 5.3. PyOCN is a multi-topology, modular, and highly parametrized on-chip network generator built in PyMTL3. The

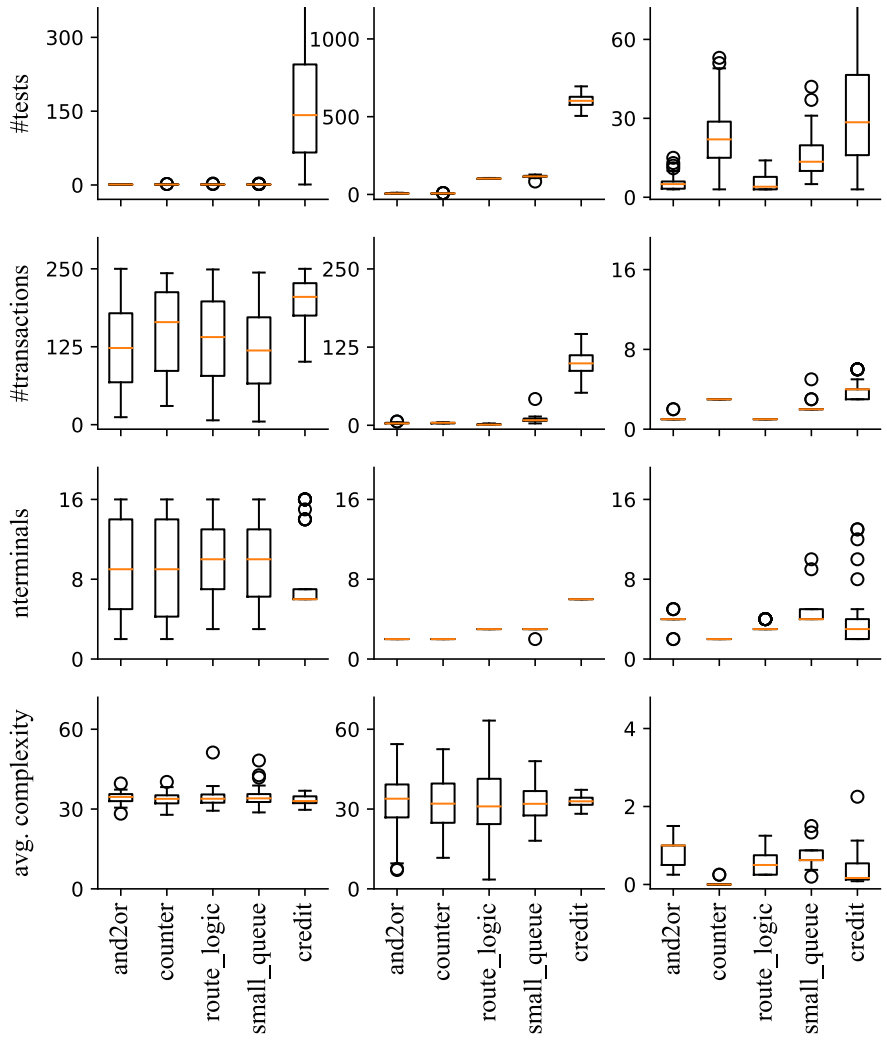
```
from hypothesis import given
from hypothesis import strategies as st

# packet_strategy is the search strategy for packets
# use @given to create composite strategies

@given( num_terminals = st.integers(2, 16),
        test_packets  = st.lists( packet_strategy() )
)
def test_ring_pyh2g( num_terminals, test_packets ):
   dut = RingNetwork( num_terminals )
   th  = TestHarness( dut, test_packets )
   run_sim( th )
```

Figure 5.2: PyH2G Strategy Example – The `@given` decorator captures `hypothesis` search strategies, and the test function can directly use the variables as normal parameters to elaborate the test harness and run tests.

example of Figure 5.3(a) shows the PyH2G test for the ring network generator. When a test case fails, `hypothesis` can simultaneously shrink the design instance and the packet sequence. We ran 50 trials for each bug to record how many tests `hypothesis` runs to find the bug and the size/complexity of the final bug reported by hypothesis. The results are shown as box-and-whisker plots in Figure 5.3(a–c).

From the results we observe that:

- PyH2G detects a failing test case quickly with a small number of test cases (similar to CRT), while IDT takes much longer to detect a failing test case. PyH2G sometimes runs slightly more test cases than CRT because `hypothesis` will first generate explicit examples to stress-test the boundary conditions before exploring values randomly. However, this also help PyH2G discover the `credit` bug more quickly than CRT.
- PyH2G produces the smaller final failing test case in terms of the number of transactions compared to CRT and IDT. This is because `hypothesis` iteratively attempts to shrink a the failing test case to produce the shortest sequence of transactions.
- PyH2G produces failing test cases with ring network instances that are significantly smaller than CRT but slightly bigger than IDT. This is because `hypothesis` may reach a local minimal failing case that cannot be simplified further to make smaller ring networks fail. In contrast, IDT always steadily increases the network size and explore many test cases for each network, which increases the chance to find failing test cases with smaller network size.



#tests: number of test cases needed to find the bug
#transactions: num. of transactions in final failing case
#nterminals: size of ring network in final failing case
avg. complexity: average complexity of all packets in a test case calculated based on value of each packet field

Bug name	Description
and2or	mistakenly put an AND instead of an OR
counter	wrong enable logic in a counter
route_logic	partially wrong routing logic in the router
small_queue	size of the router's input buffer $<$ max credit
credit	incorrect credit update logic in the router

Figure 5.3: PyOCN RingNet Generator Case Study – The box-and-whisker plots summarizes the experimental results of 50 trials for each injected bug.

- PyH2G significantly reduces the transaction complexity because the shrinking shrinks the fields of the generated messages as well. The low transaction complexity avoids unnecessary complications during the debugging phase.

5.4 PyH2P: PyH2 for Processors

PyH2P is a PyH2 framework to automatically generate random assembly instruction sequences to test processors, which makes the case for effective domain-specific random testing methodologies. Different from existing work, PyH2P is able to automatically shrink a failed long program to a minimal instruction sequence with a minimal set of architectural registers and memory addresses. It is possible to combine auto-shrinking with other sophisticated random program generators [CCSRS03] by carefully using PyH2P random strategies. PyH2P can also leverage Symbolic-QED [FUN+18] by applying QED transformations to generated random programs and performing bounded model checking to accelerate bug discovery.

5.4.1 Challenge in Testing Processors

As processors have complicated execution semantics, testing processors involve more data orchestrations than testing simple streaming hardware components or even accelerators. However, due to limited human resources, most of the open-source processor implementations only include a few directed tests, randomly generated short assembly sequences, and/or very large scale system-level tests (e.g., booting Linux). There is a critical need to create an automated random testing framework to improve the fidelity of open-source processor implementations. Recently, Google built an open-source framework RISCV-DV to test RISC-V processors [RIS]. RISCV-DV supports generating random instructions using constrained random testing. However, RISCV-DV still requires a commercial Verilog simulator that supports UVM and is specific to RISC-V. There is also work to complement RISCV-DV for RISC-V compliance negative testing [HGD20].

5.4.2 PyH2P Implementation

Different from previous random instruction generators for processors, PyH2P creates composite `hypothesis` strategies to generate random assembly programs for effective auto-shrinking. Specifically, PyH2P creates a hierarchy of strategies for arithmetic, memory, and branch instruction strategies using sub-strategies for architectural registers, memory addresses, and immediate values. PyH2P currently implements a block-based instruction generation mechanism, which first instantiates a control-flow template of branches, and then fills random instructions between branches.

```python
from hypothesis import strategies as st

# This function returns a random two-register instruction
# with two random registers

@st.composite
def inst_tworegst( draw, reg_list, inst_list ):
    reg  = st.sampled_from( reg_list )
    inst = st.sampled_from( inst_list )
    t    = draw( st.tuples( inst, reg, reg, reg ) )
    return [ Instruction( f"{l[0]} {l[1]}, {l[2]}, {l[3]}", "tworeg" ) ]
```

Figure 5.4: PyH2P Strategy Example – The `@st.composite` decorator marks the two-register instruction strategy as a composite strategy. `st.sampled_from` creates a random strategy of the given list. `st.tuples` composes multiple strategy to be drawn as a single random tuple.

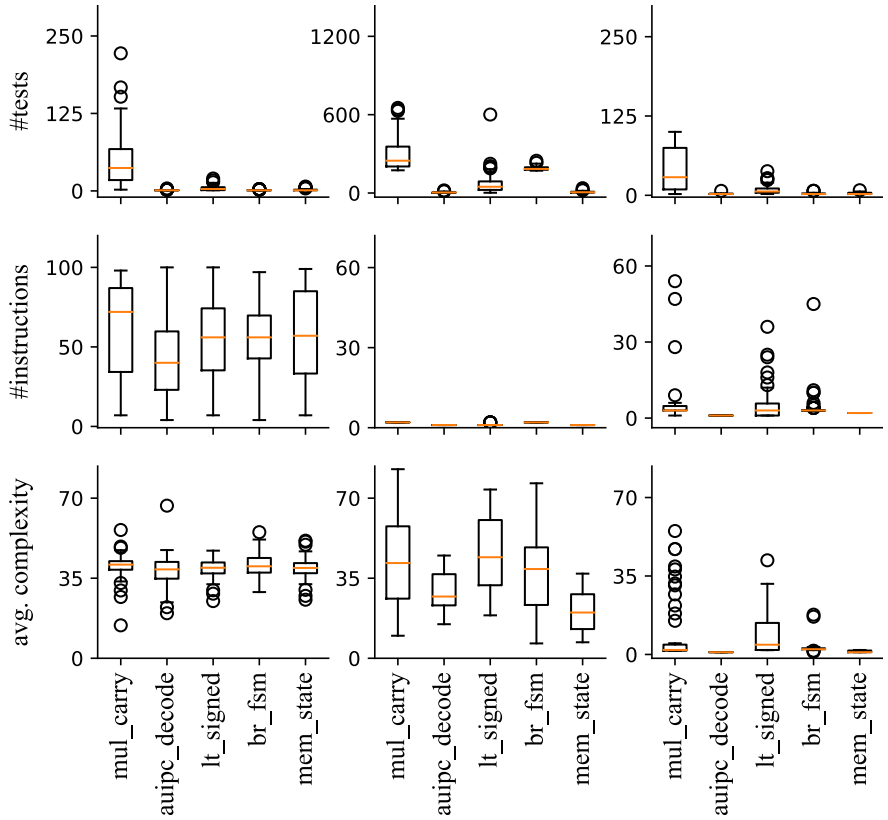
Although block-based mechanism limits the possible program space PyH2P can explore, it significantly increases the understandability of the generated test cases, which is crucial for debugging.

PyH2P also ensures that each generated assembly program has well-defined behavior across the test and reference models:

- For arithmetic instructions, PyH2P constrains the range of the immediate value strategy to avoid undefined overflow for specific instructions.
- For memory instructions, PyH2P constrains the range of the memory address strategy to the size of the provided main memory to avoid unaligned and out-of-bound memory accesses.
- For branch instructions, PyH2P first generates a sequence of branch instructions and their corresponding labels, and then randomly shuffles them to form the control-flow template. This eliminates the possibility of branch out-of-range errors. Additionally, a set of registers are dedicated to loop bounds and loop variables to avoid infinite loops.

5.4.3 Case Study: PicoRV32 Processor

We demonstrate the effectiveness of PyH2P using PicoRV32, an open-source, area-optimized RV32IMC processor implemented in Verilog. We leverage PyMTL3's Verilator support to drive the co-simulation using a PyMTL3 testbench. The imported processor is connected to a PyMTL3 cycle-level test memory which stores the assembly program generated by PyH2P. After executing the program, we extract and compare the value of PicoRV32 architectural registers and the test memory against an ISA simulator written in PyMTL3.



(a) CRT (b) IDT (c) PyH2P

#tests: number of assembly programs to discover the bug
#transactions: number of instructions in the failing program
avg. complexity: average complexity of all instructions calculated based on the registers and immediate values used

Bug name	Description
mul_carry	carry bits of carry-save adders shifted by 1
auipc_decode	a typo in the decode logic of auipc
lt_signed	a signed comparison performed w/o casting
br_fsm	incorrect state transition for branches
mem_state	the status of current load is ignored

```
1  bge   x5, x14, 0x60
2  auipc x8, 0x100
3  beq   x14, x27, 0x2C
4  xori  x23, x14, 0x6C0A
5  sltu  x23, x14, x5
6  lui   x27, 0xBD6E
7  srli  x27, x21, 0xB1C2
8  sltiu x14, x21, 0x3307
9  bltu  x14, x27, 0x64
10 sltiu x21, x5, 0x2FCF
11 // 30 insts omitted
12 mul   x14, x27, x27
```

(d) CRT Example

```
1 lui   x5, 0xC349
2 mul   x5, x5, x5
```

(e) IDT Example

```
1 lui   x1, 0x6
2 mul   x1, x1, x1
```

(f) PyH2P Example 1

```
1 lui   x1, 0x0
2 // PC=0x204 here
3 auipc x1, 0x0
4 // 0x204 * 0x204
5 mul   x1, x1, x1
```

(g) PyH2P Example 2

Figure 5.5: PicoRV32 Processor Case Study – (a)–(c) are box-and-whisker plots that show the results of each methodology. (d)–(g) show the failing cases for the mul_carry bug discovered by each methodology.

We inject five directed bugs into the Verilog code, and ran 50 trials for each methodology and bug combination. The results are shown as box-and-whisker plots in Figure 5.5(a–c). CRT generally requires a small number of tests (less than 50) to discover a bug, but the failing cases usually include more than 50 complex instructions. IDT significantly reduces the number of instructions in the failing test case, but needs significantly more cases to find the failing case. Note that IDT generates instructions of similar complexity to CRT because we have to generate random immediate values to avoid prohibitively long runtimes to find these bugs. PyH2P is able to discover the failing test case using a similar number of trials to CRT and can shrink it to a minimal case with similar length to the cases found by IDT. Moreover, PyH2P is able to shrink the immediate value so that the average instruction complexity is significantly reduced.

Figure 5.5(d–g) shows the failing cases for the `mul_carry` bug discovered by each methodology. The bug is about a misshifted bit in the multiplier, which can only be triggered by specific operands. Figure 5.5(d) is the example found by CRT with 41 instructions, 7 unique architectural registers, and large immediate values. Figure 5.5(e) shows the example found by IDT which uses only one register but a large random immediate value. This is because we only iteratively deepen the list of instructions, and have to randomize the operands to prevent prohibitly long evaluation time. Figure 5.5(f–g) include two minimal failing cases from different PyH2P runs, which are significantly simpler. The two cases are basically two local minimas of multiple runs of PyH2P testing. The first one is a two instruction sequence with a smaller operand in the first instruction than the case found by IDT. This shows the advantage of auto-shrinking in reducing value complexity. The second one has three instructions, and PyH2P basically failed to shrink it to two instructions. Specifically, the program counter starts at 0x200, but a PC of 0x204 will trigger the bug. The effect of having the first NOP instruction is to increase the program counter from 0x200 to 0x204. Because of this intricate dependency in the instruction sequence, the shrinking process ends at the three-instruction sequence.

5.5 PyH2O: PyH2 for Object-Oriented Hardware Data Structures

PyH2O is a PyH2 framework that enables using method calls to test RTL hardware components with object-oriented latency-insensitive interfaces. The key contribution of PyH2O is a novel test-

ing methodology for concurrent hardware data structures that are difficult to thoroughly test using traditional approaches. PyH2O proposes a novel simulation mechanism called *auto-ticking*, which has been implemented as a new PyMTL3 simulation pass. With merely "transaction-accurate" Python data structures as reference models, PyH2O uses the rule-based stateful testing features in `hypothesis` to perform a sequence of random method calls on both the reference model and the auto-ticking simulator of the RTL model, and then checks if the outcomes match for each method call.

5.5.1 Challenge in Testing Hardware Data Structures

Many open-source hardware blocks are designed to improve reusability by exposing well-encapsulated timing-insensitive handshake interfaces that can provide an object-oriented view of the hardware block (e.g., a hardware reorder buffer exposes three object-oriented "method" interfaces: allocate, update, and remove). However, it is very hard to perform random testing to test the behavior of concurrent hardware data structures that have multiple interfaces accepting "transactions" in the same cycle. Converting a random transaction sequence into cycle-by-cycle test vectors using traditional testing approaches requires a cycle-accurate golden model. Manually creating multi-transaction test-vectors only works for directed testing. One possible solution is to execute only one random transaction in each cycle, yet the inability to stress intra-cycle concurrent behavior hurts the quality of the tests.

We conclude that there is critical need to create a novel testing approach for object-oriented hardware using concurrent intra-cycle transactions.

5.5.2 PyH2O Implementation

PyH2O is based on method-based interfaces which are decoupled handshake interfaces with four ports: enable, ready, arguments, and return value. Essentially, setting the enable signal high after making sure the ready signal is high is equivalent to calling the corresponding ready method, checking if it returns true, and then calling the actual method. Converting an RTL method interface to a Python method involves an adapter that provides a method and a ready method to the user and sets/modifies the signals inside the adapter. PyH2O leverages Python reflection to automati-

cally wrap the RTL method interfaces with a generated top-level PyMTL3 wrapper with Python methods.

PyH2O applies the `AutoTickSimPass` to create an auto-ticking simulator for the wrapped model. Conceptually, auto-ticking is more fine-grained than the classical delta cycle approach. Auto-ticking divides the combinational logic into multiple parts based on logic related to the method interfaces. When the user calls the enhanced top-level method, not only the method but also all the logic between this method and the next method is executed. If the executed method is the last method of the cycle, the simulator advances to the first method of the next cycle. If the user skips a method in this cycle and calls another method later in the cycle or a previous method that is already skipped/called in the current cycle, the simulator ignores the in-between methods and executes all the logic until it reaches the called method. Unlike trivial one-method-per-cycle testing, this auto-ticking scheme is able to execute multiple methods in the same cycle if they are called in a specific order.

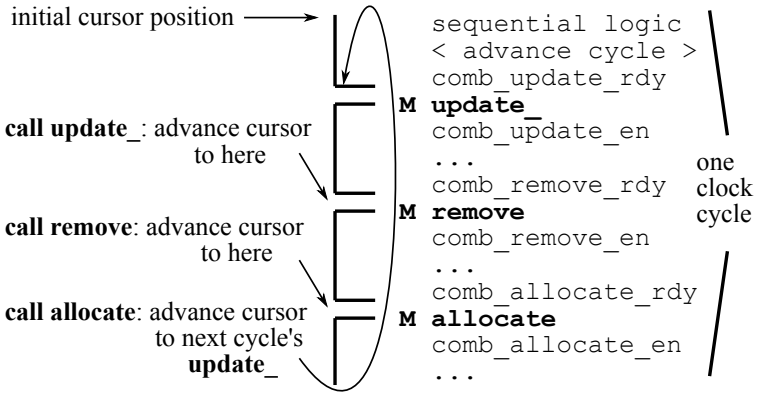
5.5.3 Case Study: Reorder Buffer Data Structure

Figure 5.6(a) shows an RTL reorder buffer implementation which exposes three method callee interfaces. `allocate` is ready if the buffer is not full. It returns the entry index and advances the tail pointer. `update_` is ready if the buffer has valid elements. It takes an index/value pair to update the buffer. `remove` is ready if the buffer head is valid and already updated, and returns the index/value pair. Note that `remove` and `allocate` can occur in the same cycle even if the reorder buffer is full, because the implementation combinationally factors whether `remove` is called into `allocate`'s ready signal. Figure 5.6(b) shows the execution schedule generated by the `AutoTickSimPass`. The auto-ticking simulator guarantees that a sequence of three method calls in the order of `update_` $<$ `remove` $<$ `allocate` will occur in the same cycle.

To show the effectiveness of PyH2O, we replace `head+1` with `head+0` in line 19 of Figure 5.6(a). This subtle bug needs at least six transactions in a specific order to trigger because it requires six transactions to allocate, update and remove two entries, but must not remove the first one and allocate the second one in the same cycle. After trying several sequences with varying length from 5 to 19, PyH2O discovers a 11-transaction failing case as shown in Figure 5.6(c). After auto-shrinking, PyH2O successfully finds one of the minimum failing case as shown in Figure 5.6(d).

```python
class ReorderBuffer( Component ):
  def construct( s, TData, num_entries ):
    TIndex  = mk_bits( clog2( num_entries ) )
    TROBMsg = mk_bitstruct( 'ROBMsg', {
      'index': TIndex, 'value': TData,
    })
    # Method-Based Callee Interfaces
    s.allocate = CalleeIfcRTL( RetType=TIndex )
    s.update_  = CalleeIfcRTL( MsgType=TROBMsg )
    s.remove   = CalleeIfcRTL( RetType=TROBMsg )
    s.head = Wire( TIndex )
    ...
    @update_ff
    def upff_head_pointer():
      if   s.reset:
        s.head <<= 0
      elif s.alloc.en & s.remove.en:
        s.head <<= s.head + 1
      elif not s.alloc.en & s.remove.en:
        s.head <<= s.head + 1 # "head + 0" bug
```

(a) PyMTL3 reorder buffer code snippet



(b) Auto-tick execution schedule for reorder buffer

```
state = ReorderBuffer_PyH2O()
state.allocate()
state.update(msg=ROBMsg(Bits2(2),Bits16(63527)))
state.allocate()
state.update(msg=ROBMsg(Bits2(3),Bits16(2091)))
state.allocate()
state.allocate()
state.update(msg=ROBMsg(Bits2(1),Bits16(62455)))
state.update(msg=ROBMsg(Bits2(0),Bits16(38580)))
state.remove()
state.alloc()
state.remove() # error: ref ready, dut not ready
```

(c) The first falsifying example found by PyH2O

```
state.allocate()
state.update(msg=ROBMsg(Bits2(0),Bits16(0)))
state.allocate()
state.remove()
state.update(msg=ROBMsg(Bits2(1),Bits16(0)))
state.remove() # error: ref ready, dut not ready
```

(d) Minimized failing case after auto-shrinking

Figure 5.6: PyH2O Case Study: Reorder Buffer – (a) shows a code snippet of the reorder buffer implementation in PyMTL3 with method-based interfaces and the injected bug annotation; (b) illustrates the auto-ticking schedule that mixes of top-level methods and the update blocks; (c–d) include the method transaction list before and after auto-shrinking.

5.6 Conclusion

This chapter has introduced PyH2, which leverages PyMTL3, `pytest`, and `hypothesis` to create a novel open-source hardware testing methodology. We believe PyH2 is an important first step towards addressing these key challenges in open-source hardware testing: (1) PyH2 is more accessible to open-source hardware designers compared to complex closed-source hardware testing methodologies; (2) PyH2G is well-suited for testing not just design instances but also design generators which are critical to the success of the open-source hardware ecosystem; (3) PyH2P can improve the random testing of open-source processor implementations compared to the more limited directed and random testing currently used in many open-source projects; and (4) PyH2O can more effectively test object-oriented hardware data structures.

CHAPTER 6
CONCLUSION

This thesis presented my work on productive and extensible hardware modeling, simulation, and verification methodologies. This thesis addressed key challenges to hardware modeling methodologies in the era of heterogeneous system-on-chips. For each of the four identified challenges, I proposed a solution to address the challenge. I also built the PyMTL3 framework that incorporates all of these solutions. Note that the proposed techniques (NIMIR, UMOC, Mamba++, PyH2) are not restricted to the specific setting in this thesis or the PyMTL3 framework; they can also inspire improvements and optimizations in other state-of-the-art hardware modeling frameworks.

6.1 Thesis Summary and Contributions

This thesis began by discussing hardware design trends in the last twenty years where the prevailing hardware platform evolved from single-core architectures to multi-core architectures and to heterogeneous system-on-chips. Creating productive hardware modeling methodologies is one of the prominent ways to reduce the non-recurring costs of building heterogeneous system-on-chips, and there have been multiple generations of hardware modeling methodologies as introduced in this thesis.

There are four key parts in state-of-the-art hardware development workflows: (1) the hardware modeling framework itself; (2) the hardware modeling abstraction; (3) simulation of the hardware models; and (4) testing/verification of the hardware models. I identify one key challenge in each part: (1) improving the flexiblity and extensibility of HGSFs; (2) unifying CL and RTL modeling to achieve high model fidelity with little effort; (3) closing the simulation performance gap in HGSFs; and (4) reducing testing/verification time for agile hardware design flows. This thesis addressed the four challenges as summarized below.

I first focused on the hardware modeling framework itself. A framework can serve as a sustainable research/engineering platform only if it is flexible and extensible. I proposed native in-memory intermediate representation (NIMIR) to address the first challege. I also presented the PyMTL3 framework, the first framework implemented under the NIMIR scheme. I discussed PyMTL3 modeling features, NIMIR implementation, and various PyMTL3 passes. To demonstrate

the flexiblity and extensibility of PyMTL3, I presented a case study on supporting delay-annotated gate-level modeling.

I then focused on the hardware modeling abstraction and proposed unified modular ordering constraints (UMOC) to unify CL and RTL modeling and achieve high model fidelity with little efforts. I implemented UMOC modeling primitives and scheduling algorithms in PyMTL3 for prototyping and evaluation. As case studies confirmed, UMOC enables high fidelity CL modeling and seamless composition of RTL and CL components.

Moving on to the simulation of the hardware models, I proposed Mamba++ techniques to close the simulation performance gap in state-of-the-art HGSFs. I used PyMTL3 as the research platform with a framework-JIT co-optimization approach. Evaluation results showed that Mamba++ techniques can improve the simulation performance by 20–100$\times$ in both pure Python and Python-Verilator co-simulation.

For testing/verification, I presented PyH2, our vision and techniques to reduce the testing/verification time for agile hardware design flows. Leveraging Python, other parts of the PyMTL3 framework, and `hypothesis`, we built the PyH2G, PyH2P, and PyH2O testing frameworks to test hardware generators, processors, and object-oriented hardware data structures, respectively. PyH2's property-based testing is able to find smaller failing cases and design instances than complete random testing and find failing cases much faster than iterative deepened testing. PyH2O is a perfect example of novel simulation mechanisms enabling more effective testing.

To reiterate the major contributions of this thesis:

- I proposed native in-memory intermediate representation (NIMIR), a novel and elegant framework architecture to improve the flexibility and extensibility of productive hardware modeling frameworks.
- I proposed unified modular ordering constraints (UMOC), a novel technique to unify CL and RTL modeling and achieve high model fidelity with little efforts.
- I proposed Mamba++, a set of novel JIT-aware HGSF design techniques and HGSF-aware JIT optimization techniques, to close the simulation performance gap in HGSFs.
- I presented PyH2, our vision and techniques to reduce the testing/verification time for agile hardware design flows. PyH2 currently includes testing methodologies for hardware generators, processors, and object-oriented hardware data structures.

- I presented PyMTL3, a novel open-source hardware generation and simulation framework. PyMTL3 incorporates all of the techniques in this thesis. In practice, PyMTL3 has been used in courses at Cornell University, research projects, and chip tapeouts in advanced technology nodes.

6.2 Future Work

Here, I list a few possible ideas to pursue in the future based on this thesis. For each research idea, I discuss the motivation, the possible research work involved, and the potential impact.

6.2.1 Making PyMTL3 and Chisel/FIRRTL Interoperate

Motivation – As Chisel/FIRRTL currently becomes the most popular circuit intermediate representation in academia and industry, there have been many open-source hardware IPs built using Chisel. The PyMTL3 framework is considered complementary to Chisel/FIRRTL: PyMTL3 focuses more on model-level view of the hardware block instead of circuit-level; and PyMTL3 focuses on smooth simulation/testing experiences. Combining the benefits of Chisel and PyMTL3 looks very appealing. Even though the designer can use black-box Verilog import of PyMTL3 which directly co-simulates the Chisel/FIRRTL generated Verilog with the PyMTL3 test harness, a white-box import of the FIRRTL model is preferred, as PyMTL3 has access to the model hierarchy and hence can transform the model using PyMTL3 passes.

Research – This research topic has two directions. One direction is to build a PyMTL3 backend in FIRRTL so that we can generate PyMTL3 code from FIRRTL. The PyMTL3 backend in FIRRTL involves duplicating the FIRRTL emitter for Verilog and changing each code generation segments to generate PyMTL3 design code. Fortunately, since FIRRTL only supports single-line assignments, PyMTL3 lambda statements can fulfill the need. There is a preliminary version of the PyMTL3 backend in FIRTL. However, there is one subtle issue to be resolved. One prominent modeling primitive in Chisel is the arithmetic type system where unsigned integer and signed integer are subtypes of Bits, and signed/unsigned integers have corresponding execution semantics. PyMTL3 only supports unsigned integers, so it might require some work in PyMTL3 or smart conversions in the backend. The other direction is to build a FIRRTL backend in PyMTL3 that

translates a subset of RTL models to FIRRTL. This involves limiting the translation of single-line lambda blocks in the FIRRTL backend.

Potential Impact – Developing a PyMTL3 backend in FIRRTL and/or developing a FIRRTL backend in PyMTL3 brings the interoperability between PyMTL3 and Chisel, which is a huge step forward for the open-source hardware community. On one hand, PyMTL3 developers obtain various IPs built in Chisel in a white-box fashion "for free", which can be used for experiments and case studies. On the other hand, FIRRTL users can generate PyMTL3 code to leverage all the PyMTL3 features such as PyMTL3 passes and PyH2 testing. With both directions done, PyMTL3 can provide a FIRRTL-in-FIRRTL-out experience.

6.2.2 Unified Scheduling for FL, CL, RTL, and Delay-Annotated GL Models

Motivation – State-of-the-art hardware development frameworks essentially focus on raising the level of abstraction to improve the productivity of users. As a result, most frameworks deploy cycle-based simulation and at most simple timing-accurate simulation such as scheduling multiple clock domains using least-common-multiple of the frequencies. In contrast, the HDLs (Verilog or VHDL) support RTL, simple GL, and timing-annotated GL models, with timing-accurate simulators. The state-of-the-art design flow is broken down to two phases: the first phase includes FL, CL, and RTL modeling in the productive hardware modeling framework, and the second phase include low-level RTL, GL, and delay-annotated GL models in HDL simulators. To further improve the productivity in the first phase, there is a need for a novel unified scheduling scheme to simulate FL, CL, RTL, and delay-annotated GL models altogether in the hardware modeling framework.

Research – This research topic involves two steps. First, the current existing delay-annotated GL modeling scheme as studied in Section 2.4 is not compatible with the general UMOC scheduling scheme. A unified scheme that combines UMOC and delay-annotated GL simulation is a potential starting point. This step can further be broken down into several steps of investigation: GL and UMOC parts are separated; GL is at the top and UMOC is one clock domain; or UMOC is at the top and GL is intra-cycle. Second, the new scheduling scheme should be more complicated than the current UMOC scheme and probably cannot directly reuse Mamba++'s hierarchical static scheduling to achieve high simulation performance. Optimizing the simulation performance of the unified UMOC/GL scheme may involve some existing Mamba++ techniques or new techniques.

Potential Impact – A high-simulation-performance unified scheduling scheme for FL/CL/RTL/GL models can bring significant benefits in chip prototyping. If done right, this work has the potential to be a new open-source EDA tool.

6.2.3 Exploring Fully Offloaded Simulation to Verilator Inside PyMTL3

Motivation – Section 4.9 discussed the case where the 5-stage pipeline processor without caches has *lower* simulation performance than the larger 5-stage pipeline processor with caches due to the difference in the test memory activity in Python. Analysis confirms that the C++ part of the simulation is significantly faster than the Python part. Specifically, performing simulation in pure C++ has the potential to bring another 2–20$\times$ of speedup to the game as suggested by the Verilator simulation performance in Chapter 4. It becomes very appealing to offload more simulation to Verilator during PyMTL3/Verilator co-simulation for even higher simulation performance.

Research – This research topic involves two steps. First, in order to explore such offloaded simulation, the test memory, test sources, test sinks need to be made translatable to Verilog (but no need to be synthesizable). This requires some efforts in carefully constructing PyMTL3 testing components in the mindset of ROMs and registers instead of Pythonic integers and lists. As a side note, it is also interesting to see how those translatable test memory/source/sink can be synthesized into hardware. The second step is to design a novel co-simulation interface between PyMTL3 and Verilator. This is because PyMTL3 still needs to correctly drive the co-simulation and know when the Verilator simulation ends or when some error is thrown. This interface/callback scheme needs to be lightweight and asynchronous between PyMTL3 and Verilator to avoid bringing in other unnecessary overheads.

Potential Impact – Successful translation and synthesis of these testing components may open up opportunities in productive FPGA methodologies which can help FPGA prototyping and chip bring-up. Then, if the offloaded PyMTL3/Verilator co-simulation scheme takes little effort to set up, it can potentially increase designers' interest in using PyMTL3 to build huge RTL blocks. Writing RTL design in PyMTL3 and setting up ultra-fast virtual prototyping using Verilator with little efforts in other parts of the test harness seem to be superior to any existing approach.

6.2.4 Exploring PyMTL3/Synopsys VCS Co-simulation

Motivation – Although the `VerilogTBGenPass` provides a handy way to create cycle-by-cycle Verilog test benches, the generated test benches are not feasible to drive billion-cycle Verilog simulations due to two reasons: (1) the test case files are too huge since they record all the value changes for every cycle; and (2) it is not realistic to even simulate a billion cycle in PyMTL3 with the `VerilogTBGenPass` activated. PyMTL3 already supports PyMTL3/Verilator black-box co-simulation. However, when it comes to serious silicon prototyping, Verilator as a community-maintained open-source Verilog simulator falls short of the industry standard. The most used commercial Verilog simulator is Synopsys VCS. Additionally, VCS is a four-state simulator, while Verilator is only two-state. In conclusion, black-box cosimulation with VCS becomes a very appealing option over Verilator.

Research – There are simple example of leveraging VCS slave mode [vb0] to compile VCS into a C++ library. The `VcsInit()` and `VcsSimUntil()` seems like a tick function to advance the time stamp. It may require some efforts to figure out how to instrument the values in the actual Verilog design. Then, it might involve building fake Verilog wrappers to enable the C++ wrapper generated by the PyMTL3 VCS import pass to exchange value with the Verilog world. Essentially, the key idea is to mimic how we did for Verilator import to build Python/C++ (and potentially Verilog) wrappers to enable value exchange between PyMTL3 and VCS in each cycle.

Potential Impact – PyMTL3/VCS black-box co-simulation will make post place-and-route or signoff gate-level simulation much easier. The same PyMTL3 test bench can be used to drive the GL simulation. The envisioned automated flow (Verilog translation, ASIC flow and black-box 4-state VCS simulation) is essentially an enhanced version of the current Verilator co-simulation flow.

6.2.5 Exploring the Spectrum Between Constructive and Transformative Hardware Design

Motivation – The PyMTL3 transform passes open up vast opportunities to avoid temporarily rewriting code in many different files and reverting them later. This is especially useful in the iterative debugging process. Moreover, consider a physical/logical mismatch that PyMTL3 can fix. Two designers are following good engineering practice. One implements the processor which encapsulates the processor and the caches in a single module. The other one implements the

on-chip network model which encapsulates all the routers in a single module. Composing them together does not lead to any issue in simulation, but when it comes to the physical design step, it becomes impossible to point the ASIC tool to a tile that consists of one processor and one router. To perform 2D floorplanning, the designers are required to create a new module and put one processor and one router inside it, which breaks the modularity. A PyMTL3 transform pass can "grab" those routers out from the network component and put it into the processor/cache tile. Then the modified PyMTL3 model can be translated to Verilog, and then the tiles can be floorplanned by ASIC tools. Such usage of transform passes is fundamentally different from the traditional constructive hardware design where hardware components are fully declared before elaboration. Taking one step forward, what if the whole model is constructed using a transform pass?

Research – The research topic involves three steps. First, hand-crafting a few fully transformative hardware designs such as a CGRA accelerator is a good starting point. This involves carefully constructing a mini program that starts with an empty PyMTL3 component and only uses PyMTL3 API calls to add and connect components/signals. Then, slowly add various components to the empty component and remove part of the mini program until it reaches a sweet spot of succinct component description and succinct mini program using regular loops. Finally and hopefully, there is some interesting insights in the transformative hardware design approach and some useful ideas from studying the sweet spots.

Potential Impact – This spectrum between constructive and transformative hardware designs is very profound and worth investigation. The discussion above is only tip of the iceberg. Note that this topic can also use FIRRTL/Chisel as the infrastructure, but the model-level view provided by PyMTL3 may be the key enabler of such research.

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.

[ABC+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A System for Large-Scale Machine Learning. *Symp. on Operating System Design and Implementation (OSDI)*, Nov 2016.

[ACF+19] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem, and et al. Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project. *Design Automation Conf. (DAC)*, Jun 2019.

[AKPJ09] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2009.

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

[ARKK13] A. Annamalai, R. Rodrigues, I. Koren, and S. Kundu. An Opportunistic Prediction-Based Thread Scheduling to Maximize Throughput/Watt in AMPs. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2013.

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

[BCC+17] D. Bradford, S. Chinthamani, J. Corbal, A. Hassan, K. Janik, and N. Ali. Knights Mill: New Intel Processor for Machine Learning. *Symp. on High Performance Chips (Hot Chips)*, Aug 2017.

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

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

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

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

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

[Bol12] J. Bolaria. Xeon Phi Targets Supercomputers. *Microprocessor Report (MPR)*, Sep 2012.

[boo11] BookSim Interconnection Network Simulator. Online Webpage, 2011 (accessed Dec 19, 2011). `https://nocs.stanford.edu/cgi-bin/trac.cgi/wiki/Resources/BookSim`.

[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.

[BYF+09] A. Bakhoda, G. L. Yuan, W. W. L. Func, H. Wond, and T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2009.

[CCA+11] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, Feb 2011.

[CCSRS03] F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero. Fully Automatic Test Program Generation for Microprocessor Cores. *Design, Automation, and Test in Europe (DATE)*, Mar 2003.

[CH00] K. Claessen and J. Hughes. QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs. *Int'l Conf. on Functional Programming (ICFP)*, Sep 2000.

[CKES17] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss - An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits (JSSC)*, 52(1):127–138, Jan 2017.

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

[CM08] P. Coussy and A. Morawiec, editors. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

[CMJ+18] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. *OSDI*, Oct 2018.

[CML08] A. Chattopadhyay, H. Meyr, and R. Leupers. LISA: A Uniform ADL for Embedded Processor Modeling, Implementation, and Software Toolsuite Generation. In *Processor description languages*, pages 95–132. Elsevier, 2008.

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

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

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

[DPR96] C. Dawson, S. Pattanam, and D. Roberts. The Verilog Procedural Interface for the Verilog Hardware Description Language. *IEEE International Verilog HDL Conference*, 1996.

[EBA+11] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2011.

[FUN+18] M. R. Fadiheh, J. Urdahl, S. S. Nuthakki, S. Mitra, C. Barrett, D. Stoffel, and W. Kunz. Symbolic Quick Error Detection Using Symbolic Initial State for Pre-Silicon Verification. *Design, Automation, and Test in Europe (DATE)*, 2018.

[GALP18] N. Ganjehloo, V. Akella, and J. Lowe-Power. Integrating Cycle Accurate Chisel Models with gem5's System Simulation, 2018.

[GHN+12] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. DySER: Unifying Functionality and Parallelism Specialization for Energy Efficient Computing. *IEEE Micro*, 33(5), Sep/Oct 2012.

[Gre11] P. Greenhalgh. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *EE Times*, Oct 2011.

[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.

[HGD20] V. Herdt, D. Große, and R. Drechsler. Closing the RISC-V Compliance Gap: Looking from the Negative Testing Side. *Design Automation Conf. (DAC)*, Jun 2020.

[HGG+99] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration Through Compiler/Simulator Retargetability. *Design, Automation, and Test in Europe (DATE)*, Mar 1999.

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

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

[iee21] P1800 - Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. Online Webpage, 2021 (accessed May 10, 2021). `https://standards.ieee.org/project/1800.html`.

[IKL+17] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2017.

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

[JBM+13] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally. A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2013.

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

[JOP+20] S. Jiang, Y. Ou, P. Pan, K. Cheng, Y. Zhang, and C. Batten. PyH2: Using PyMTL3 to Create Productive and Open-Source Hardware Testing Methodologies. *IEEE Design & Test*, 40(4):58–66, Jul/Aug 2020.

[JOPB21] S. Jiang, Y. Ou, P. Pan, and C. Batten. UMOC: Unified Modular Ordering Constraints to Unify Cycle- and Register-Transfer-Level Modeling. *Design Automation Conf. (DAC)*, Dec 2021.

[JPOB20] S. Jiang, P. Pan, Y. Ou, and C. Batten. PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification. *IEEE Micro*, 40(4):58–66, Jul/Aug 2020.

[JTB18] S. Jiang, C. Torng, and C. Batten. An Open-Source Python-Based Hardware Generation, Simulation, and Verification Framework. *Workshop on Open-Source EDA Technology*, Nov 2018.

[KDK+11] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, Sep/Oct 2011.

[KFJ+03] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2003.

[KJJ+20] Y. D. Kim, W. Jeong, L. Jung, D. Shin, J. G. Song, J. Song, H. Kwon, J. Lee, J. Jung, M. Kang, et al. A 7nm High-Performance and Energy-Efficient Mobile Application Processor with Tri-Cluster CPUs and a Sparsity-Aware NPU. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2020.

[KJT+17] J. Kim, S. Jiang, C. Torng, M. Wang, S. Srinath, B. Ilbeyi, K. Al-Hawaj, and C. Batten. Using Intra-Core Loop-Task Accelerators to Improve the Productivity and Performance of Task-Based Parallel Programs. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2017.

[KTMH07] T. H. Khan, S. Tahar, O. A. Mohamed, and A. Habibi. Automatic Generation of Systemc Transactors from Graphical FSM. *Int'l Conf. on Microelectronics (ICM)*, Dec 2007.

[KTR+04] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[LA04] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Int'l Symp. on Code Generation and Optimization (CGO)*, Mar 2004.

[LAS+09] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2009.

[LFSZ17] T. Liang, L. Feng, S. Sinha, and W. Zhang. PAAS: A System Level Simulator for Heterogeneous Computing Architectures. *Int'l Conf. on Field Programmable Logic (FPL)*, Sep 2017.

[Lie84] K. J. Lieberherr. Towards a Standard Hardware Description Language. *Design Automation Conf. (DAC)*, Jun 1984.

[LK09] J. Lee and N. S. Kim. Optimizing Throughput of Power- and Thermal-Constrained Multicore Processors using DVFS and Per-Core Power-Gating. *Design Automation Conf. (DAC)*, Jul 2009.

[LKK+18] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2018.

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

[LSC+10] M. Lis, K. S. Shim, M. H. Cho, P. Ren, O. Khan, and S. Devadas. DARSIM: A Parallel Cycle-Level NoC Simulator. *Workshop on Modeling, Benchmarking and Simulation (MOBS)*, Jun 2010.

[LWC+16] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic, et al. An agile approach to building RISC-V microprocessors. *IEEE Micro*, 36(2):8–20, 2016.

[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.

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

[MD20] D. R. MacIver and A. F. Donaldson. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer (Tool Insights Paper). *European Conference on Object-Oriented Programming*, Nov 2020.

[MFN+17] M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, J. Balkind, A. Lavrov, M. Shahrad, S. Payne, and D. Wentzlaff. Piton: A Manycore Processor for Multitenant Clouds. *IEEE Micro*, Mar 2017.

[MHDmoc19] D. R. MacIver, Z. Hatfield-Dodds, and many other contributors. Hypothesis: A New Approach to Property-Based Testing. *Journal of Open-Source Software (JOSS)*, 4(43), Nov 2019.

[mig] Migen: A Python Toolbox For Building Complex Digital Hardware. `https://m-labs.hk/gateware.html`.

[MMB+18] C. Mattarei, M. Mann, C. Barrett, R. G. Daly, D. Huff, and P. Hanrahan. CoSA: Integrated Verification for Agile Hardware Design. *Int'l Conf. on Formal Methods in Computer Aided Design (FMCAD)*, Oct 2018.

[MMG+20] O. Matthews, A. Manocha, D. Giri, M. Orenes-Vera, E. Tureci, T. Sorensen, T. J. Ham, J. L. Aragón, L. P. Carloni, and M. Martonosi. MosaicSim: A Lightweight, Modular Simulator for Heterogeneous Systems. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Aug 2020.

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

[MRR12] M. McCool, A. D. Robinson, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012.

[myh21] MyHDL: From Python to Silicon. Online Webpage, 2021 (accessed May 15, 2021). `http://www.myhdl.org`.

[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.

[NM15] M. Naylor and S. Moore. A Generic Synthesisable Test Bench. *Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, Sep 2015.

[ope08] OpenMP Application Program Interface. OpenMP Architecture Review Board, 2008. `http://www.openmp.org/mp-documents/spec30.pdf`.

[PACG11] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS-x86: A QEMU-Based Micro-Architectural and Systems Simulator for x86 Multicore Processors. *Design Automation Conf. (DAC)*, Jun 2011.

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

[Ped20] V. A. Pedroni. *Circuit Design with VHDL.* The MIT Press, 2020.

[PFKM06] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo Graph Embedding: Mapping Applications onto Coarse-Grained Reconfigurable Architectures. *Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Oct 2006.

[PGM+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *CoRR arXiv:1912.01703*, 2019.

[PMH+21] P. Paternoster, A. Maki, A. Hernandez, M. Grossman, M. Lau, D. Sutherland, and A. Mathad. XBOX Series X: A Next-Generation Gaming Console SoC. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2021.

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

[pyp21] PyPI: The Python Package Index. Online Webpage, 2021 (accessed May 10, 2021). `https://pypi.org/`.

[PZK+17] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun. Plasticine: A Reconfigurable Architecture For Parallel Paterns. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[RCBJ11] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters (CAL)*, 10(1):16–19, 2011.

[Rei07] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[RIS] RISCV-DV. `https://github.com/google/riscv-dv`.

[RZAH+19] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, A. Rao, T. Ajayi, J. Puscar, S. Dai, R. Zhao, D. Richmond, Z. Zhang, I. Galton, C. Batten, M. B. Taylor, and R. G. Dreslinski. A 1.4 GHz 695 Giga RISC-V Inst/s 496-core Manycore Processor with Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS. *Symp. on Very Large-Scale Integration Circuits (VLSIC)*, Jun 2019.

[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.

[SBM+19] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. B. S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli. MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2019.

[SDF06] S. Sutherland, S. Davidmann, and P. Flake. *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Springer Science & Business Media, 2006.

[SGC+16] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46, Mar/Apr 2016.

[Sha81] M. Sharir. A Strong-Connectivity Algorithm and Its Applications in Data Flow Analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[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.

[Tar71] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *Annual Symp. on Switching and Automata Theory (SWAT)*, Oct 1971.

[Tay13] M. B. Taylor. A Landscape of the New Dark Silicon Design Regime. *IEEE Micro*, 33(5):8–19, Sep/Oct 2013.

[TM08] D. Thomas and P. Moorby. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[TOJ+19] C. Tan, Y. Ou, S. Jiang, P. Pan, C. Torng, S. Agwa, and C. Batten. PyOCN: A Unified Framework for Modeling, Testing, and Evaluating On-Chip Networks. *Int'l Conf. on Computer Design (ICCD)*, Nov 2019.

[vb0] vb000. VCS Slave Mode. `https://github.com/vb000/vcs-slave-mode/tree/master`.

[ver21] Verilator. Online Webpage, 2021 (accessed May 15, 2021). `http://www.veripool.org/wiki/verilator`.

[VSS+20] R. Venkatasubramanian, D. Steiss, G. Shurtz, T. Anderson, K. Chirca, R. Santhanagopal, N. Nandan, A. Reghunath, H. Sanghvi, D. Wu, et al. A 16nm 3.5 B+ Transistor> 14TOPS 2-to-10W Multicore SoC Platform for Automotive and Embedded Applications with Integrated Safety MCU, 512b Vector VLIW DSP, Embedded Vision and Imaging Acceleration. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2020.

[WJM08] W. Wolf, A. A. Jerraya, and G. Martin. Multiprocessor system-on-chip (MPSoC) technology. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(10):1701–1713, 2008.

[You07] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2007.