Improving FPGA Simulation Capacity with Automatic Resource Multi-Threading

by

Albert Forte Magyar

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Krste Asanović, Co-chair
Adjunct Assistant Professor Jonathan Richard Bachrach, Co-chair
Professor Sanjit Seshia
Associate Professor Kyle Steinfeld

Spring 2021

Improving FPGA Simulation Capacity with Automatic Resource Multi-Threading

Abstract

Improving FPGA Simulation Capacity with Automatic Resource Multi-Threading

by

Albert Forte Magyar

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Krste Asanović, Co-chair

Adjunct Assistant Professor Jonathan Richard Bachrach, Co-chair

Modern system-on-a-chip (SoC) development is a highly complex process that spans multiple levels of design abstraction and cross-cutting requirements. With a rapidly evolving ecosystem of domain-specific accelerators and wide design spaces to search, the ability to rapidly evaluate potential chip designs has never been more important. In the modern chip-design landscape, field-programmable gate arrays (FPGAs) play a critical role in delivering this simulation capability due to their unique ability to emulate concrete, register-transfer level (RTL) designs at speeds sufficient to run real applications spanning trillions of cycles of simulated target-design execution. However, the use of FPGAs for logic emulation presents challenges, including the perennial difficulty of effectively mapping large target designs to the finite resources of a given FPGA platform.

To help address this challenge, this dissertation presents a novel approach to manage these limitations through the use of automatic resource-efficiency optimizations that reduce the number of FPGA resources required to faithfully implement cycle-accurate emulators of large chips, all without requiring the tedious manual effort and complexity of previous FPGA-optimized simulation techniques. By substituting target-design memories with logic-intensive read and write ports for resource-efficient, cycle-accurate models that serially access FPGA memory primitives, Golden Gate simulators can avoid the disproportionate impact of FPGA-hostile memory design patterns on simulators of high-performance processor cores. Drawing inspiration from software simulators and specialized emulators, where common code may be repeatedly executed to model an arbitrary number of copies of a given block, I also introduce an automatic instance-threading optimization, through which the logic resources required to simulate a given module may be shared across multiple instances, radically reducing their collective footprint.

To support the use of these optimizations across a broad array of user designs, they are integrated as contributions to Golden Gate, an extensible compiler that translates RTL designs into cycle-accurate FPGA simulators as part of the open-source FireSim FPGA simulation framework. By structuring simulators as modular dataflow networks, Golden Gate provides the flexibility to compose the two optimizations along with the ability to combine them with software co-simulation or other advanced simulation features. To evaluate the performance of the optimizations and to validate the optimizing compiler stack, these techniques are applied to two input designs: a general-purpose SoC with multiple out-of-order cores and a domain-specific accelerator with multiple systolic array co-processors. In each case, finite programmable logic resources limit the maximum number of cores–and therefore the size of the system–that can effectively be simulated on a simulation platform consisting of cloud-hosted Xilinx VU9P FPGAs. However, by enabling optimizations in Golden Gate through simple compiler directives, the same FPGA platform was able to support configurations of each system with an eight-fold increase in core count relative to the baseline, providing the ability to simulate sixteen out-of-order cores or eight accelerator cores at high speed, with deterministic, cycle-accurate results. Ultimately, this significant increase in per-FPGA capability broadens the utility of commodity FPGAs in simulating ever-growing chips, while the convenience of automatic compiler optimization helps support designer productivity in a rapidly accelerating hardware ecosystem.

Contents

List of Figures

List of Tables

Glossary of Terms

Abbreviations

ASIC	Application-Specific Integrated Circuit
FAME	FPGA Architecture Model Execution
FPGA	Field-Programmable Gate Array
LI-BDN	Latency-Insensitive Bounded Dataflow Network
LTL	Linear Temporal Logic
MIDAS	Modeling Infrastructure for Debugging and Simulation
RTL	Register Transfer Level

Definitions

Chisel	An open-source hardware description language embedded in Scala
FAME-1	A simulator that may execute one target cycle across multiple host cycles
FAME-5	A simulator where a host resource may represent n copies in the target
FIRRTL	Flexible Intermediate Representation for RTL: the output of Chisel
Host	The system of hardware & software resources used to implement a simulator
LIME	Latency-Insensitive Model Equivalence: a checker for FAME-1 simulators
RISC-V	An open standard instruction set architecture
Target	The system & environment that are simulated by an architectural simulator
RAMP	Research Accelerator for Multiple Processors: a many-core FPGA simulator

Acknowledgments

Though there are many uncertainties in the PhD process, I have been certain throughout my time at Berkeley that I am very lucky to have an amazing group of colleagues. Their mentorship led me to pursue graduate school, and their support enabled me to finish.

- I am deeply grateful to my co-advisor Jonathan Bachrach for always believing in me. From offering a student in class a job to seven years of collaboration, you've been a welcome source of optimism and forward-looking vision.
- Without a common vision, the fantastic collaboration in the Berkeley Architecture Research group would not be possible. My co-advisor Krste Asanović has helped foster a true team dynamic, which has been the foundation of lasting relationships with other graduate students. Krste has always been a reliable source of great ideas and a dedicated teacher.
- I would like to particularly thank Sanjit Seshia for helping to shepherd our EE219C project and for his detailed feedback on the ICCAD publication on Golden Gate.
- Unfortunately, I am saddened to hear of the untimely passing of Pramod Subramanyan. His patient and enthusiastic help as an instructor for EE219C was the greatest enabler in learning more about model checking and UCLID5.
- As my first mentor in the ASPIRE Lab, Palmer Dabbelt taught me more than I thought I wanted to know about computers. He is a one-of-a-kind friend who is always willing to share.
- My long-term collaborator Adam Izraelevitz has always been a source of inspiration. From letting me sleep on his couch, to working together on class projects, to seven years of Chisel meetings, he's always made everything a bit more fun. Most of all, I appreciate his ability to bring positivity any situation.
- Through many projects (and even more project acronyms), I have always enjoyed collaborating with Jack Koenig. The quality of his insights are matched only by his ability to never take himself too seriously.
- I know I am not alone in thanking Andrew Waterman for his consistent willingness to spend time discussing ideas, even long after his own graduation from Berkeley. His dedication to teamwork and to continual improvement of mature projects like Chisel continue to have lasting impact at Berkeley and beyond.
- For many years, Eric Love has been both a reliable friend and a tireless driver of a close, friendly research group environment. Outside of work, he has been a part of many of the great memories of my time at Berkeley. In these isolating times, I will always look back fondly on our many Zoom calls, especially those with our friends Nathan Pemberton and Jordan Kellerstrass.

- Last, but certainly not least, without the help of David Biancolin, I would never have made it through the latter half of my PhD. His collaboration in so many aspects of my research helped make this entire dissertation possible. More importantly, he has been a tireless source of advising and has always helped me keep a clearer vision of where my work is headed. Outside of work, his humor and shared interests have made grad school much more enjoyable.

Outside of work, I would like to thank my family for a lifetime of support. My parents, along with my sister Lydia and brother Derek, have always made me feel as though I have the capability to take on anything. I know that grad school can be a test of patience for any parents, but I never felt pressure to pursue a more immediate or practical path. Finally, most of all, I would like to thank my fiancée Orianna for always listening when I needed it most, for being the best Powerpoint editor I know, and for helping show me by example that finishing grad school is in fact possible.

Chapter 1

Introduction

As the semiconductor industry ventures further into the twilight of transistor scaling, there is broad consensus that improvements in computing performance and energy efficiency must come from innovations above the transistor in the computing stack. This comes at a time when there are myriad emerging applications, in domains like AI, virtual and augmented reality, and the Internet of Things, that depend on the availability of higher-performance, more energy-efficient computing systems. As a result, system architects have turned to specialization: in modern SoCs, application cores increasingly yield their area to specialized accelerators [36]. However, this specialization begets complexity that makes these systems more difficult to build, verify, and program. This drives up the non-recurring engineering (NRE) costs of developing new chips, making custom silicon inaccessible to all but high-volume markets.

The lack of an affordable full-system simulation technology that is both fast and accurate is one key driver of these NRE costs. A simulator that is too slow cannot exercise bugs that manifest deep into execution and is thus unusable for software development. However, a faster, less detailed simulator may differ too greatly from the actual silicon to exhibit the same bugs and performance pathologies, precluding effective pre-silicon verification and validation.

Among the leading solutions for digital logic simulation, Field-Programmable Gate Arrays (FPGAs) have long been used for prototyping and emulation of ASICs in both industry [55] and academia [91, 49]. While FPGAs have great potential as a commercial-off-the-shelf technology that offers radical speedups over software simulation, no current FPGA-based system offers the ideal combination of simulation speed, capacity, affordability, and ease of use. Direct *FPGA prototypes* are affordable and fast, but require the user to manually model the external environment of the device and to invest significant effort to meet resource constraints. Commercial *emulation platforms* offer automated scaling to larger designs, but suffer from high cost of entry and take a large performance hit when partitioning designs across many FPGAs. While manually resource-optimized simulators such as RAMP Gold [83] present large increases in per-FPGA capacity, this approach lacks the flexibility to model arbitrary target designs. When contrasted with the simplicity of software simulation,

it is clear that FPGAs provide the potential for large speedups at the expense of flexibility. In this work, we hypothesize that FPGA simulation can be improved by employing abstractions inspired by software simulators and specialized emulation processors.

In general, the problem of mapping a design to an FPGA-based emulation platform centers around translating register-transfer level (RTL) designs to direct FPGA implementations, which imposes a fundamental capacity constraint: every component in the target design must have a corresponding component in the FPGA implementation. This is very much in contrast to either a software simulator or an advanced emulation platform[12], which use the notion of serial execution to extend the capacity of a finite set of hardware resources such as an application processor or specialized emulation chip. Traditionally, FPGA capacity limitations have been addressed via manual optimization of the *target design* (the design under simulation) to better suit the host FPGA platform, or by partitioning the design across multiple FPGAs. While these techniques are valuable and often necessary, they are grounded in the limitations that arise from the direct translation of a target RTL design into an FPGA implementation. Though this direct translation is convenient, it is unnecessarily restrictive.

This dissertation presents an alternative approach: automatic construction of resource-optimized FPGA simulators by introducing a *simulator compiler* to the simulation workflow. In contrast with traditional FPGA tools, such a compiler is explicitly aware of the notion that it is creating a simulator, rather than an implementation. This distinction provides the freedom for the compiler to transform components of the RTL design into simulation models that do not rely on an equivalent implementation at the RTL level; rather, these models are hardware implementations of a specification that implies cycle-accurate simulation of the target design.

As a concrete implementation of this concept, this dissertation introduces automatic resource optimizations in Golden Gate, a simulator compiler framework that accepts target RTL designs and produces RTL implementations of cycle-accurate simulators. By introducing this extra level of abstraction, Golden Gate decouples the cycle-by-cycle execution of the FPGA simulator from the cycle-by-cycle forward progress of simulated target execution time. While other simulation frameworks have used the notion of independent FPGA and simulation time to reconcile issues arising from the I/O boundary or to support co-simulation of software-hosted models, Golden Gate is designed from the start to support optimizations such as time-multiplexing of FPGA resources to enable flexible tradeoffs of simulation throughput and resource efficiency. In contrast to previous work in static time-multiplexing of FPGA resources, Golden Gate employs the Latency-Insensitive Bounded Dataflow Network (LI-BDN) formalism to decompose the simulator into sub-components, each of which may be independently and automatically optimized. This structure allows Golden Gate to support a broad class of optimizations that improve resource utilization by implementing FPGA-hostile structures over multiple cycles, while the LI-BDN formalism ensures that the simulator still produces bit- and cycle-exact results.

This work focuses on two composable optimizations that provide significant increases in per-FPGA capacity, allowing much larger systems to be simulated on a single FPGA.

The first optimization replaces highly ported memories in the target design with multi-cycle simulation models that rely on serialized accesses to efficient underlying FPGA memory resources. While this optimization is applicable only to very specific blocks in the target design, it provides a significant benefit when applied to RTL implementations of multi-ported register files in out-of-order processors. In addition, a second optimization focuses on reducing the effective footprint of a broad swath of the design: the large footprint of the combinational logic contained in repeated instances of large blocks, which are a signature feature of modern multi-core systems. When viewing an FPGA simulator as a hardware implementation of a simulation specification, rather than as a direct implementation of the target design, it is clear that there is considerable redundancy; therefore, the *multi-threading* optimization allows Golden Gate to serialize the simulation of multiple instances. This optimization provides a radical increase in capacity at the expense of extending the latency required to simulate a single cycle of the set of instances.

Finally, while these optimizations provide significant benefit in isolation, the advantage of the structured approach of Golden Gate to generating simulators allows the two optimizations to compose. While the multi-threading optimization radically reduces the resource utilization of the combinational logic in multiple instances, it does nothing to reduce the footprint of FPGA-hostile memories. However, when composed with the memory optimization, the resulting simulator provides relative capacity increases exceeding the product of the two optimizations' individual benefits. Furthermore, as in other domains, the use of multiple threads of execution is effective at hiding latency increases elsewhere in the system; in practice, this allows the composed simulator to deliver performance similar to the theoretical maximum of a simulator employing the instance multi-threading optimization alone.

In addition to the two optimizations, this work involves substantial contributions to the hardware compiler infrastructure that underpins Golden Gate. In addition to several components of the Golden Gate software implementation, the development of these optimizations included various efforts to validate the behavior of the simulator implementations. One component of this validation process is LIME, the Latency-Insensitive Model Equivalence checker. This dissertation presents LIME and other infrastructural improvements as examples of how improved software support can improve productivity in developing computer-aided design (CAD) tools for simulation and other applications in digital system design.

1.1 Previous Publication, Collaboration, and Funding

Portions of this work were published at the 2019 International Conference on Computer-Aided Design as "Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes." This project is designed to integrate with the FireSim[49] project; specifically, it replaces MIDAS, the hardware compiler that allows FireSim to transform target designs to add the ability to "pause" the advance of simulated time. While MIDAS relied on a fairly simple transformation, Golden Gate introduces an optimizing compiler

to allow the simulator to trade off time for reduced utilization of scarce host FPGA resources.

The baseline, non-optimizing compiler framework for Golden Gate was developed in collaboration with David Biancolin. While this dissertation focuses on the research contribution of enhancing FPGA capacity with automatic resource optimization, most of the engineering effort is associated with co-developing this original software framework. Furthermore, the pattern described in Section 4.5 was also developed through the collaborative integration of the multi-ported memory models with the rest of the Golden Gate compiler. Finally, the lime verification tool was originally developed as a class project for EE219C with David Biancolin and Jack Koenig.

The information, data, or work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0000849. Research was partially funded by ADEPT Lab industrial sponsors and affiliates Intel, Apple, Futurewei, Google, and Seagate, and supported by gifts provided by Amazon Web Services and Xilinx.

The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Chapter 2

Background

The work presented in this dissertation draws on a long background of prior work in FPGA simulation. Many academic and industrial projects have made notable advances in resource-efficient use of commodity FPGAs, some of which have laid the groundwork for the optimizations presented in Chapters 5 and 6. Furthermore, we rely on a broad assortment of open-source projects to provide infrastructure for generating parameterized target designs, developing hardware compiler transforms, and targeting cloud-hosted FPGA host platforms.

2.1 Prior Work in FPGA Emulation

Pre-silicon evaluation of ASICs has long been a core application for FPGAs [25]. While this takes many forms, including prototyping, emulation, and hardware-accelerated simulation, each involves mapping a *target* system (the device being simulated) onto a *host* system that includes one or more FPGAs.

2.1.1 FPGA Prototyping

Direct FPGA prototypes, where designs are directly mapped onto FPGA fabric, are a common way to enable pre-silicon software development and functional validation [1]. At a high level, an FPGA prototype implementation is produced by providing the RTL design of a chip as an input to a standard FPGA development suite, such as Xilinx Vivado or Intel Quartus Prime. By appropriately mapping the I/O pins of the FPGA and designing a suitable board, such an implementation can be used as a near-direct replacement for the chip under *in-circuit emulation*.

Ideally, this would be a push-button flow, but in reality, multiple hurdles often necessitate the labor-intensive development of an "FPGA version" of the design:

1. *Device capacity*: nontrivial ASICs must be partitioned across multiple FPGAs at the expense of slower execution rates, longer compile times, and more expensive host platforms [37].

2. *Resource conversions*: ASIC power, reset, and clocking structures do not map directly to the host FPGA and must be replaced [1, 38].

3. *I/O modeling*: I/O devices and environment models may not map well to the fabric, necessitating adapters for in-situ prototyping. One recurring example of this issue is the need to slow down external I/O to match the reduced speed of an FPGA prototype.

With a traditional FPGA prototype, the burden of overcoming these hurdles is left to the user.

2.1.2 Commercial Emulation Systems

Commercial FPGA-based emulation systems generally consist of a custom hardware platform, along with a set of software tools to streamline the partitioning and I/O modeling problems [55]. These tools build on advances in inter-chip routing [51, 38] and time-multiplexing of pins [4] to reduce the speed and productivity overhead of using multi-FPGA host platforms. Furthermore, they may offer *transactional emulation* mechanisms for interfacing with I/O models that are co-simulated in a software environment [43, 56], which can resolve the issue of I/O speed matching by gating the clock in the target design to wait for software. However, these features come at a price: large monetary cost of entry and slowdowns due to partitioning.

In addition to FPGA-based systems, multiple commercial emulation platforms rely on custom emulation chips as an underlying implementation substrate, including the IBM Yorktown Simulation Engine [73], Cadence Palladium [12] and Mentor Graphics Veloce [86]. While these systems employ similar multi-chip partitioning strategies to large FPGA emulators, each chip relies on application-specific emulation accelerators that execute an instruction stream, rather than an intrinsically parallel FPGA implementation. This key distinction brings many of the advantages associated with programmability, including the flexibility to accept larger programs (and therefore simulate more logic), along with the ability to implement arbitrarily complex behaviors, including gate-level simulation and non-RTL constructs of modeling-oriented HDLs. Furthermore, since these products' toolchains effectively generate *software* images for emulation processors, they have much faster compile times than those of large multi-FPGA emulators, which may be measured in multiple days. However, as evidenced by the continuing popularity of FPGA-based solutions, these systems are not without significant drawbacks: they come at an even higher cost that is prohibitive for all but the largest chip-building operations, and the flexibility of programmability comes at the cost of lower emulation throughput. Though academic efforts such as Malibu [35] and Cyclist [5] have attempted to provide low-cost alternatives, specialized emulation hardware remains the domain of large CAD tool vendors.

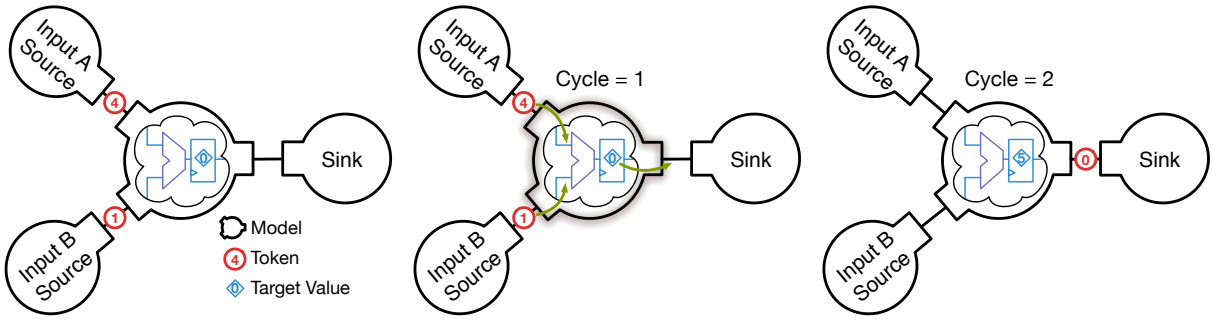

Figure 2.1: A 32-bit adder model and environment simulating a single cycle of target time.

2.1.3 Decoupled FPGA-Accelerated Simulators

While computer architecture research has long relied on software simulators in lieu of complete RTL implementations, the Research Accelerator for Multiple Processors (RAMP) [91] project aimed to use FPGAs to increase the speed and fidelity of microarchitectural simulations of many-core systems. This cross-university initiative led to the development of numerous FPGA simulators; in contrast with commercial logic emulation systems that model the behavior of concrete RTL designs, these simulators were generally designed from the ground up to optimally model a particular target system or class of systems. By exploiting introspection of the behavior of the target design in the design of the host implementation, these simulators were able to overcome some of the intrinsic the resource constraints faced by FPGA prototypes. In particular, some RAMP simulators such as HAsim [71] and RAMP Gold [83] used optimized RTL timing models to model FPGA-hostile structures like multi-ported RAMs over multiple FPGA cycles. This *host-target decoupling*, the ability to simulate one target clock cycle over a variable number of FPGA-host clock cycles, is the hallmark of these *decoupled simulators*.

To support host-target decoupling, the target machine can be simulated as a synchronous dataflow graph [63] of *models*; we give an example in Figure 2.1. To simulate one target cycle, a model dequeues one *token* from each of its input ports and enqueues a token into each of its output ports. The simplest RTL implementation of a model waits for all of its input tokens to be available and all output ports to be ready before executing; this is a direct application of Carloni et al. [14]. Simulation models that properly implement this formalism tolerate latency on the arrival of tokens and may take variable number of host cycles to compute their outputs. This makes it possible to apply these optimizations without changing the target's RTL behavior.

An important measure of decoupled simulator performance is the FPGA-Cycle-To-Model-Cycle Ratio (FMR) [72]: the average number of FPGA cycles elapsed per simulated target cycle over a full simulation. The simulation rate of a decoupled simulator can thus be given as $f_{\text{FPGA}}/\text{FMR}$. In contrast, a direct FPGA prototype by definition has FMR = 1 and a

resulting simulation rate of f_{FPGA}[1].

To date, such optimized simulators have seen little adoption, as their specialized timing models are difficult to design, optimize, and validate—which for nontrivial models may be far more complex than simply implementing the target design.

2.1.4 The FAME Simulator Taxonomy

Developed to help navigate the proliferation of academic FPGA simulators developed under the umbrella of the RAMP project, the FPGA Architectural Modeling and Execution (FAME) [84] taxonomy outlined three primary dimensions of the simulator design space. The binary presence or absence of three main features—decoupling, derivation from a high-level abstract description, and multi-threading—are used to encode a number in increasing order of significance. While this schema presents convenient labels, it belies some of the subtlety in systems that may potentially combine all three.

Decoupling

The lowest-order bit of the FAME taxonomy denotes whether a simulator is decoupled or direct. As a result, a simulator with decoupling alone is represented as the 3-bit binary number `001`. For convenience, this is generally referred to as a "FAME-1" simulator.

However, this simple property does not capture any relative *degree* of decoupling in a particular simulator. In particular, given that many decoupled simulators are based upon an implementation split across one or more FPGAs and various software components, it is worth examining whether there is decoupling within the FPGA component. A simple decoupled simulator might simply decouple a monolithic FPGA component from the rest of the simulator, implying that the portion of the system simulated on the FPGA advances in time in unison. In contrast, more elaborate simulators might feature decoupled interfaces among disparate FPGA-based simulation models.

Abstractly Specified Models

The next bit—in the 2^1 place—encodes whether the simulator uses a full RTL design (zero) or a higher-level description (one) to specify the behavior of the target. It is generally well understood that the use of higher-level specifications—and their correspondingly weaker constraints—allow more algorithmic freedom to synthesize optimal implementations. While RTL provides a relatively strict specification for a system, an abstract model might elide microarchitectural details and adhere only to an architectural specification. This relaxation in constraints could lead to a more resource-efficient implementation.

While this broader solution space could theoretically be exploited by a compiler, there is no guarantee that an automated flow from an abstract specification to an FPGA implemen-

[1] In some partitioned FPGA prototypes, "FMR" is actually a fixed number greater than one to allow for serialization-deserialization of target signals that span multiple FPGAs.

tation would necessarily outperform a traditional RTL-to-FPGA flow. Indeed, one need only examine the long and incremental road to viability of modern High-Level Synthesis (HLS) tools—which map behavioral algorithms to hardware implementations—to understand that abstract specifications do not always lend themselves to producing effective results in hardware systems.

In light of these challenges, real-world efficiency gains from abstract FAME models have generally come from careful human effort and engineering. A common pattern is the separation of an abstractly specified simulator into two parts: a simplified *functional model* that lacks microarchitectural detail, and a reduced-cost *timing model* that, while including fine-grained microarchitectural detail, may elide the significant fraction of the hardware dedicated to computing functional results. This timing-functionality split has generally been achieved through careful hand design of simulation models in prototypical systems such as FAST [20] and HAsim [71].

This reliance on handwritten design highlights an interesting question about abstract simulation models: are they more user-friendly than full RTL simulators? Though intrinsically subjective, it depends on multiple concrete properties of the entire simulation task, including the design cycle of the target system, the desired accuracy, the degree of variation in the target design space, and the time budget available for simulator development.

During early design exploration, the use of an abstract model avoids the need to develop a fully fledged RTL implementation of the target. However, since there is no general way to synthesize an implementation realizing the potential efficiency improvements of an abstract model, this is offset by the need to instead develop an RTL implementation of the model itself, where the need to explicitly consider the behavior of the host simulator adds additional challenge. However, while simulators relying on split timing and functional models have seen limited deployment due to their implementation complexity, well-designed abstract models can offer considerable flexibility, with new features being prototyped in hours, rather than days [84].

In contrast, when designs are further along in their life cycle, there is likely already a full RTL specification. Therefore, the use of abstract models would represent duplicated effort, and the ideal simulator compiler would operate on the ASIC-ready RTL. Though this increases the barrier to modeling extreme changes to the the target system, it also obviates the need to validate such changes in an abstract simulator. While abstract models may suffer from the same tradeoffs of flexibility and validation that have dogged high-level software architectural simulators [84], the use of RTL specifications ensures cycle accuracy by construction. Furthermore, advanced tools for writing hardware *generators*—discussed in Section 2.2—can help bridge the productivity gap in exploring large design spaces.

Multi-threading

The highest-order bit in the classification reflects the use of *threading*, a means by which the FPGA resource utilization of a simulator can be radically reduced at the cost of simulation throughput. While a direct mapping of an RTL design to an FPGA will result in a fully

parallel implementation that is theoretically capable of simulating one target cycle in a single host FPGA cycle, a threaded simulator will rely on serialization of the implementation relative to this parallel baseline. As discussed at length in Chapter 6, this is typically achieved by using a single underlying datapath to simulate multiple identical instances of a particular block or subset of the target design. While this reduces the peak throughput of the simulator by a degree equal to the number of replicated instances, scheduling logic that interleaves the different threads of simulation corresponding with the different instances can achieve good effective throughput by hiding other latencies, including that of modeling decoupled I/O.

Combining Features

As discussed above, while the FAME taxonomy relies on binary flags to represent the features employed by a simulator, not all combinations are practical. The decoupled simulators including these features fell into three primary categories: FAME-1, FAME-3, and FAME-7.

Employing decoupling alone, a FAME-1 simulator relies on the ability to "pause" the progress of simulated target time to model the I/O of a chip. Though this presupposes a division of the simulator into separate *system* and *environment* components, this is a natural fit for such simulators, as the system is specified as an RTL design. While FAME-1 simulators have appeared primarily in academic research, including the Green Flash [92] and MIDAS [54] (discussed further in 2.1.5), this paradigm is also employed by the commercial SCE-MI [43] standard for integrating FPGA-hosted simulations with software-defined modeling of external I/O.

In general, any abstractly specified component of a simulator would preclude a direct mapping of target to FPGA implementation, so this feature is naturally coupled with some degree of decoupling. Therefore, the use of abstract models in a decoupled simulator would be encoded as `011`, and such a simulator would colloquially be called a "FAME-3" system. Similarly, the use of multi-threading generally implies a decoupling of host FPGA clock cycles and simulated time. While a "FAME-5" simulator would provide a threaded, decoupled implementation of a concrete RTL specification, the extant simulators that inspired the original FAME taxonomy tended to couple the use of threading with the use of abstract specifications. In particular, advanced "FAME-7" simulators such as HAsim and RAMP Gold tended to rely on threading of independent functional and timing models for a highly optimized FPGA resource footprint.

Hybrid Simulators and Model Granularity

While the FAME levels provide an intuitive classification for FPGA simulators, they lack the nuance to describe the full space of decoupled simulators. For example, while the use of abstract models would promote a decoupled simulator to FAME-3, it is crucial to note that this does not preclude the use of concrete RTL specifications for some components of the target design. Indeed, abstract and RTL specifications each provide distinct and complementary advantages; a practical *hybrid simulator* might transform the RTL implementation

the majority of a target SoC into a decoupled simulator while incorporating abstract models of difficult-to-implement, expensive components such as DRAM interfaces.

2.1.5 FireSim and Golden Gate

While decoupled FAME simulation offers advantages in capacity, flexibility, and co-simulation capability, the engineering effort associated with developing a decoupled simulator has proven to be a significant barrier to adoption. Furthermore, the effort expended on developing abstractly specified models is often highly coupled to a particular target architecture—a hindrance to adopting agile hardware design methodologies. To address this gap, the FireSim [49] project aims to strike a balance between the generality of prototyping and the desirable features of decoupled simulation. To this end, it occupies a useful niche in the FAME space: co-simulating software models of off-chip components with a decoupled SoC simulator derived from full ASIC RTL. Furthermore, with an extreme focus on versatility and ease of use, it aims to provide a "batteries included" experience to a broad variety of users.

FireSim is a multi-layered framework that includes not only FPGA simulation tools for modern SoCs, but also a high-quality reference platform for computer architecture research. It includes configurable generators for Rocket-Chip-based systems (see Section 2.4), a collection of programmable accelerators, tools for generating Linux-based target software workloads, and co-simulated network simulation models to enable deterministic, coordinated simulation of multiple SoCs in a cluster- or datacenter-scale network. When simulating multiple SoC nodes, a networked FireSim simulation can be viewed as the composition of SoC-level units, each responsible for the cycle-accurate simulation of a single node. This layer involves a contrast with previous work in decoupling, as the compiler infrastructure of FireSim automatically generates a decoupled simulation model from the RTL implementation of the target design. Though the simulator may rely on composition with hand-written or software-based models, this "core" model is automatically *transformed* from the input ASIC RTL; in the simplest case, this becomes a FAME-1 component of an overarching FAME-3 simulator.

To preserve modularity in this complex workflow, the task of transforming the portion of the target design specified via concrete RTL is performed by a dedicated *simulator compiler*. While many components of the FireSim ecosystem are in some way specialized for the existing space of target designs, this simulator compiler is more general, operating on a nearly arbitrary input design. At the time of this writing, there have been two generations of simulator compilers included with FireSim: MIDAS and Golden Gate.

MIDAS

Throughout its initial development and release, FireSim relied on a non-optimizing compiler, MIDAS [54], to convert the portion of the target design specified via RTL to a monolithic decoupled model. The model transformation performed by MIDAS was fairly straightforward: it automatically added handshaking interfaces to the I/O boundary and added enable

signals to state updates that could "pause" forward progress, allowing the model to stall while it waits for tokens [52]. This technique enables co-simulation of network interfaces to model networks of thousands of target machines [49] and FPGA-accelerated modeling of the external DRAM interfaces of the target ASIC [8]. While this resembles the clock-gating approach used to support transactional emulation [43], the flexible decoupled interface with the target simplifies instrumentation to support power modeling and debugging features [54, 53]. However, while MIDAS effectively added FAME-1 decoupling, the ASIC RTL used within the model is largely unchanged, yielding the same resource utilization challenges as FPGA prototypes. Furthermore, this approach assumed the use of both a single clock domain and exclusively synchronous reset, restricting its applicability to simple systems not entirely representative of modern SoCs.

Golden Gate

In order to address the limitations of MIDAS, recent releases of FireSim have relied on Golden Gate, an optimizing simulator compiler for realistic SoCs [68]. Golden Gate relies on a higher degree of decoupling, allowing the portion of the target design specified via RTL to be decomposed into multiple simulator "blocks," with each independently compiled to a decoupled implementation and potentially optimized. This modular structure allows the compiler to be extended to include new features, and the work presented in this dissertation is implemented as part of Golden Gate. Chapters 3 and 4 provide an introduction to the overall structure of Golden Gate; this common infrastructure was implemented as a joint collaboration to enable both optimizations and enhanced support for target design features to be added to FireSim. The optimizations described in subsequent chapters build upon this baseline and are incorporated into the latest releases of Golden Gate.

2.2 Chisel: A Modern Language for Hardware

While the hardware design industry generally relies on the Verilog [42] and VHDL [40] hardware description languages (HDLs) to express register-transfer level (RTL) designs, these dated languages continue to present productivity hurdles for designers. In particular, while each has proven to be an effective foundation for modern ASIC flows built around logic synthesis, their low-level semantics are a strict superset of synthesizable behaviors, allowing for deeply non-intuitive mismatches with designer expectations. Furthermore, despite the excessive generality of their low-level operations, both languages have limited support for parameterization, hindering design reuse.

In many ways, the pitfalls of HDL semantics for logic design are a natural consequence of their development: neither VHDL nor Verilog was originally intended as a means to develop concrete hardware implementations; instead, they were intended to describe the behavior of a system in a modeling or simulation environment. This mismatch has led to perennial issues where expectations do not match the semantics of the language: misguided inference

of latch-based state by synthesis tools [27], four-state logic [82], and complicated ordering of statements [34] have dogged digital designers throughout the modern era of relying on Verilog-based synthesis tools for RTL design. In response, a broad industry of linters and other EDA tools has emerged to help guard the gaps between user expectations and Verilog semantics [79]; while this has proven to be useful in practice, it is an imperfect solution that also introduces extra complexity and arcane restrictions.

Given these qualities of popular HDLs, designers are faced with the paradoxical realization that such languages are simultaneously far too expressive for strict RTL semantics but insufficiently expressive to reuse implementations across common parameterization patterns. For example, while Verilog supports `generate` blocks for static replication of RTL templates, it lacks support for parameterizing the set of I/O signals of a module or polymorphically selecting which of a set of different child modules to instantiate within a parent. To circumvent these limitations, tools such as Genesis2 [80] employ a strategy similar to a C preprocessor to embed fragments of Verilog or VHDL within an enclosing program—or *generator*—in a scripting language such as Perl. When the script is executed, it assembles the fragments into a valid HDL specification; this process can employ dynamically varying execution paths, string interpolation, and rich parameter sets to allow a wide range of different designs to be produced from a single generator. This paradigm is more generally an example of *staged meta-programming*, where the execution of one program leads to the generation of another [13]. However, this approach suffers from two primary flaws: it does little to address the potential for mismatches with RTL semantics, and given the disparate semantics of the stages, it offers no guarantees that a given generator output will even be valid HDL code.

In contrast to these Perl-based preprocessing flows for Verilog, the Chisel [6] project provides a safer, more principled approach for meta-programming digital hardware generators. Rather than employing a second scripting language to assemble fragments of an existing HDL, Chisel is a *hardware construction language* (HCL) embedded in the Scala programming language [81]. A Chisel generator uses a simple core API to manipulate elements that correspond with RTL signals; while this allows the enclosing Scala program to comprise a highly parameterized generator that dynamically constructs varying circuits through its execution, the simplicity of the API guarantees that the resulting circuit is valid. Indeed, since this API contains only strict RTL behaviors, it is possible to say that any Chisel-generated circuit is *synthesizable by construction*.

Given its emphasis on predictable behavior, Chisel has naturally lent itself to CAD research. As discussed in Section 2.3, the generated circuits are represented via a portable intermediate representation, FIRRTL, which serves as a foundation for the work presented in this dissertation. Though FIRRTL is not intrinsically tied to Chisel and can be produced by other user-level tools, Chisel serves as the primary design language for both the target designs used in our experimental evaluations (introduced in Section 2.4) and for numerous instances of simulation-specific hardware in the Golden Gate implementation.

2.3 The FIRRTL Hardware Compiler Framework

Introduced as part of the compiler infrastructure for Version 3.0 of Chisel, FIRRTL [47], the Flexible Intermediate Representation for RTL, is an intermediate representation for digital circuit designs. Inspired by the success of LLVM [60], an Intermediate Representation (IR) from the software compiler world, the open-source FIRRTL project provides both a specification and a reference compiler implementation. Beyond its use in the Chisel ecosystem, FIRRTL also provides an ideal infrastructure for writing tools that manipulate digital circuits; since it is "language-agnostic" with respect to the frontend, user-facing design language, it allows tool developers to focus on the core functionality of CAD or simulation tools.

2.3.1 FIRRTL Intermediate Representation

The FIRRTL representation (IR) is designed to capture the range of essential features for modern digital circuits at the register-transfer level (RTL). In contrast with a general-purpose HDL such as IEEE Verilog [42] or SystemVerilog [41], the design of FIRRTL is specialized for writing transformations that operate at the RTL level, which provides several notable benefits to the implementer of complex tools like Golden Gate.

1. Unlike Verilog, which is capable of modeling many subtly different forms of concurrent execution, FIRRTL limits the semantics of assignment statements to RTL behavior.

2. As a direct consequence of the "RTL-only" restriction, all FIRRTL circuits are synthesizable by construction.

3. Since FIRRTL is designed to integrate with powerful front-end languages like Chisel, it lacks complex features like functions, loops, and generate statements.

4. FIRRTL provides explicit wire and register declarations that intrinsically restrict corresponding assignments to be combinational or synchronous, respectively.

5. First-class memories include port and behavior definitions, which avoids the challenges of inference and errors with multi-dimensional array accesses.

While an alternative to handling Verilog and all its complexities would be to rely on a specialized netlist format or structural subset of Verilog that disallows procedures, tasks, and functions, there is also a significant productivity advantage associated with using FIRRTL. In contrast to a pure netlist, FIRRTL includes convenience features such as composite types and conditional assignments. With these small additions, FIRRTL represents a happy medium between an HDL and a netlist that is particularly amenable to writing tools such as Golden Gate that involve not only transforming a circuit but modifying its RTL behavior in a programmatic fashion.

2.3.2 The Reference FIRRTL Compiler

While FIRRTL IR is effectively a language that is specified in a reference manual [65], the FIRRTL compiler infrastructure includes a default implementation that is structured as an extensible Scala framework, allowing the reference compiler to be reused and extended in whole or in part. The core unit of reuse in a FIRRTL flow is a *transform*, which is a functional transformation from an input circuit to an output circuit.

2.4 The Rocket Chip Generator

Rocket Chip [3] is an open-source project that provides a configurable generator to combine multiple processor implementations, coherent caches, accelerators, and peripherals to produce custom SoCs. It is industry-proven, and has served as the basis for numerous academic ASIC tapeouts in modern process technologies [77]. By combining a lengthy silicon track record, modern architectural features, extreme customization, and permissive open-source licensing, it has proven to be a valuable target design for the FireSim project. Larger, more feature-rich configurations may provide multiple cores, multiple levels of caches, page-based virtual memory (with TLBs to cache translations), and support for running Linux. Furthermore, Rocket Chip is tightly integrated with the Chisel language and therefore enables straightforward experimentation with FIRRTL-based compiler tools such as Golden Gate. Therefore, the work presented in this dissertation is evaluated against a set of target designs based on the Rocket Chip generator. While these targets rely on a common generator, the generated SoCs span a wide range of performance, power, and area (PPA) by varying parameters such as type and number of cores. In particular, we examine designs based on the in-order Rocket core and the high-performance, out-of-order BOOM core [16].

2.4.1 Rocket

In its default distribution, Rocket Chip is based on Rocket, a traditional five-stage in-order pipeline. As with the top-level SoC generator, Rocket is highly parameterizable, with support for both 32- and 64-bit RISC-V ISAs along with all standard extensions. At the microarchitectural level, the bypassing of data from older instructions that have not yet committed their architectural results to newer, dependent instructions can be enabled to improve performance or disabled to save resources. Efficient hardware support for IEEE 754 floating point operations can be selectively added, as can variable-latency units for integer multiply and divide. While Rocket issues instructions in program order, these long-latency operations can be improved by enabling a scoreboard to allow out-of-order completion.

As Rocket was the first silicon-proven implementation of the RISC-V ISA, it has been repeatedly validated in both academic and industrial systems. Furthermore, it represents a sensible combination of performance and efficiency for many applications. Therefore, it remains a mainstay of RISC-V SoCs, while also providing a mature benchmark for tools in the Chisel and FIRRTL ecosystem.

2.4.2 BOOM

BOOM, the Berkeley Out-of-Order Machine, is an open-source out-of-order, superscalar implementation of the RISC-V ISA. It is organized around explicit register renaming and a unified physical register file; in this respect, it is similar to notable out-of-order RISC processors such as the MIPS R10000 [95] or the DEC Alpha 21264 [50]. Like Rocket, BOOM is implemented as a core generator using Chisel; however, it extends this parameterization to reflect the broader design space of out-of-order cores. In addition to the sizes of discrete-capacity structures like register files and reorder buffers, BOOM provides the novel capability to vary the maximum degree of instruction-level parallelism allowed at various points in the pipeline. By appropriately choosing the fetch, decode, and issue widths, along with the mix of execution units, it is possible to generate instances ranging from an efficient two-issue core to a large, four-issue core comparable in performance to a mobile application processor. Furthermore, BOOM offers a collection of parameterizable branch prediction algorithms; users may even define their own, providing a powerful tool for computer architecture research.

Though BOOM provides unique flexibility, it is constrained by the need to provide high-quality, performant, physically realizable implementations of various design points. Since selection of these design points presents its own challenges, the BOOM repository includes several pre-defined default configurations spanning various levels of PPA; these "sane defaults" provide good starting points for research and realistic benchmarks for hardware design tools. Furthermore, as with any large open-source project, BOOM has gone through multiple revisions. To illustrate the gradual evolution in complexity—and performance—over time, a high-level pipeline diagram for each version is shown in Figure 2.2. The BOOM v2 release [15] focused on physical design considerations based on experiments in mapping the core to a modern 28nm process technology, which led to a successful tapeout. The current BOOM release, known as SonicBOOM [96], offers significantly improved performance, with instructions-per-cycle (IPC) metrics that are competitive with some comparably sized commercial cores on industry-standard benchmarks.

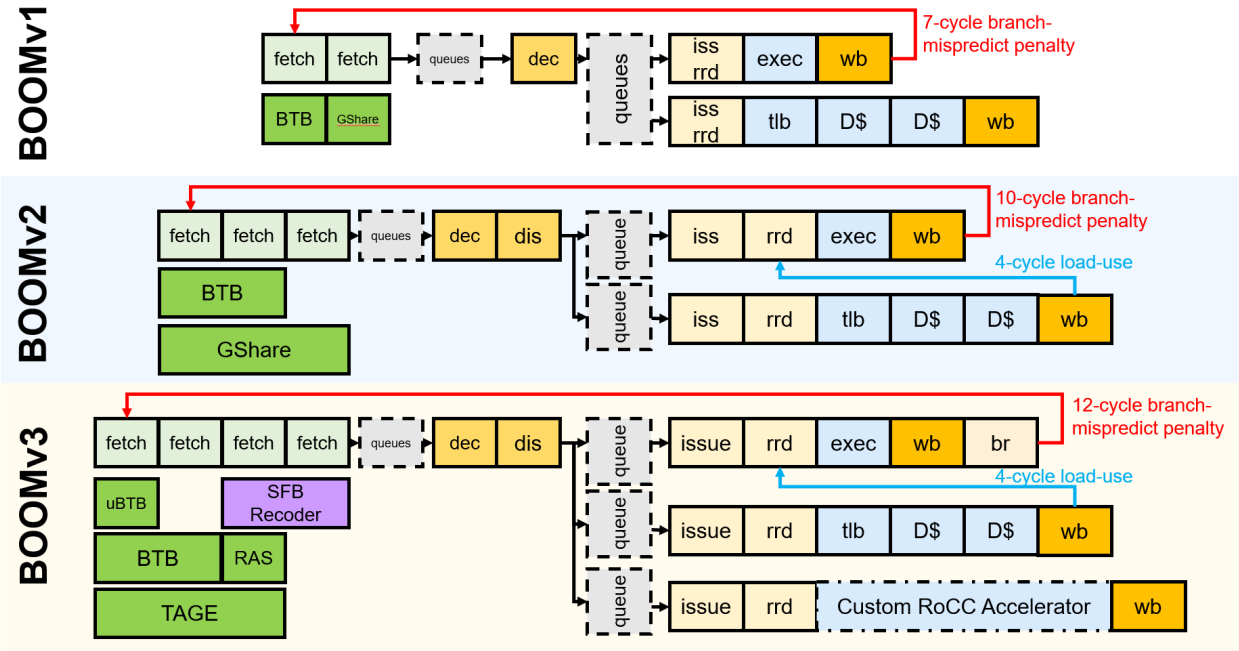

Figure 2.2: *(Figure used with permission from Jerry Zhao [96])* A set of pipeline diagrams representing the microarchitectures of the three major BOOM releases. The SonicBOOM release represents a significant step forward in performance through increased instructions-per-clock (IPC) over BOOMv2.

Chapter 3

Dataflow Simulation with Golden Gate

In the broad landscape of existing simulation FPGA simulation projects, the presence of *decoupling* has been used to delineate advanced simulators from those that rely on directly mapping input RTL to equivalent FPGA implementations. However, the definition of decoupling is largely operational: a decoupled simulator is one that may use multiple clock cycles of host FPGA execution to simulate a single cycle of target system behavior. In this chapter, we explore the nuances associated with decoupled simulators, particularly hybrid or *compositional simulators* that employ a higher degree of decoupling to enable automatic optimization of the resulting FPGA implementations. Finally, drawing upon previous work, we outline a framework for robustly constructing compositional simulators from target RTL designs.

3.1 All Simulators are Hybrid Simulators

As discussed in Section 2.1.4, advanced decoupled simulators may often apply different simulator implementation strategies to model different subsets of the the target system. These *hybrid simulators* might combine abstract models of some components, such as DRAM or PCIE interfaces, with a simulation of the remainder of the SoC that is derived from an RTL specification.

However, even the most basic simulators generally employ some form of hybrid design; when viewing a simulation as a "closed world," even a direct FPGA prototype will generally rely on some software functionality on a general-purpose host processor. Most practical FPGA-based simulators include at least some software components hosted on a general-purpose processor. As typified in the SCE-MI standard for software co-modeling, these often serve to model I/O or other components not implemented in the RTL specification for the device under test.

In this work, we focus on hybrid simulators that not only support co-simulation, but

also divide the task of simulating the RTL specification of the target design across a network of communicating *simulation models*. Each component of this network may simulate a corresponding partition of the target with an implementation that falls under a different level of the FAME taxonomy, and the resulting hybrid simulator will involve many internal, asynchronous communication interfaces that lead to a very high degree of decoupling.

3.1.1 Refining the Notion of Decoupling

Though the FAME taxonomy presents a dichotomy between FPGA prototype and decoupled simulators where the former requires a one-to-one relationship between target and host clock and the latter does not, this operational definition makes it difficult to classify many practical FPGA simulators. FPGA prototypes often involve multiple clock domains, including ones that do not directly represent a target clock. Large multi-FPGA simulators often use fast SERDES to serialize the exchange of signal values across the boundaries of the partitions; this technique to circumvent pin-count limitations spreads one element of simulating a particular target cycle over multiple periods of a high-frequency host clock. Indeed, even double-pumping of block RAMs to model highly ported memories exploits the ability to reason about the FPGA clock as a feature of the host platform. However, simulators employing these techniques alone do not generally fit with the explicit decoupling found in FAME simulators.

With this in mind, it is clear that simulators may possess varying degrees of decoupling. Arguably one of the most important features of FAME simulators like FAST, HAsim, ProtoFlex, and RAMP Gold is not merely dividing the work of simulating a target cycle across multiple host cycles, but allowing the implementation to tolerate *variable* latency. This is enabled by the use of asynchronous handshaking interfaces to exchange data among simulation models, allowing for true decoupling of their execution. Furthermore, such "globally asynchronous" simulators vary in the degree to which simulation is distributed across multiple decoupled components; the most advanced simulators like RAMP Gold blend components with single-cycle, fixed multi-cycle, and variable latencies [83].

This space of different simulation techniques is explored in some detail by the work on A-Port Networks [72], which provide an asynchronous protocol for distributing decoupled simulators across multiple FPGA-based simulation models. In addition to direct prototypes, in which there is no distinction between the FPGA and target clocks, the authors provide a taxonomy of four other classes of simulators.

- Unit-delay simulation synchronously advances the target design by one clock cycle after a static number of clock cycles. Techniques such as double-pumping of block RAMs and the use of SERDES for inter-chip wires fall under this category.
- Dynamic barrier simulation uses a centralized controller to manage host time, dynamically advancing target time after a variable number of FPGA clock cycles.

- A-Port networks distribute the simulation across a network of actors and omits a centralized, synchronized value for simulation time. By relying on decoupled interfaces, independent models may "slip" forward or backward in time relative to others. The messages exchanged within the network each represent the value of some target design variable(s) in the target machine at a particular moment in time that is implicitly encoded by their FIFO ordering.

- Chandy-Misra-Bryant simulators [17, 10] rely on a protocol that includes explicit representations of the time at which an event occurs or a signal takes a particular value. While this yields algorithmic advances by allowing the simulator to "fast-forward," cycle-accurate RTL simulation has a relatively high density of meaningful "events," and the implementation overhead is prohibitive for FPGA simulators.

While these classifications capture many of the distinctions among FAME simulators, A-Port networks are not the only protocol for implementing distributed, asynchronously communicating simulators. Therefore, we consider A-Port-based models to be one example of the broader class of FAME simulators with distributed models communicating over asynchronous channels. By introducing a high degree of internal decoupling, these architectures are ideal for hybrid simulators that blend levels of abstraction and incorporate multiple optimizations.

3.1.2 Incorporating RTL-Specified Models in Hybrid Simulators

While FireSim includes multiple abstractly specified models, the behavior of the vast majority of the target design is specified through ASIC-oriented RTL implementations. Though the use of abstract models carries numerous potential benefits, including rapid design space exploration and highly optimal simulator implementations, simulating concrete RTL provides a higher degree of confidence in the validity of the the results. Furthermore, while architectural simulation can be useful for early experiments, FireSim can be used for performance validation of designs that are close to tapeout.

Within FireSim, the Golden Gate compiler is responsible for translating the RTL specification for this subset of the target design into a decoupled simulator capable of interfacing with the remaining co-simulated models; therefore, while a full FireSim simulator is partially abstract, the optimizations presented in this work are applicable to RTL simulation. This highlights a further difficulty of precisely labeling hybrid simulators: the granularity at which different features are mixed within a single simulator, along with the complementary benefits of full RTL and abstracted simulation make Golden Gate more difficult to classify. Given the presence of abstract models, the incorporation of the multi-threading optimization discussed in Chapter 6 would classify the resulting FireSim instance as a FAME-7 simulator. However, since the optimization operates only on the RTL-specified portion of the simulator, we refer to it as a "FAME-5 compiler."

3.1.3 Hybrid Simulators and Optimization

The ability to automatically optimize the utilization of FPGA resources by trading off space and throughput has proven to be one of the most compelling advantages of decoupled simulators [84]. While it is theoretically possible to develop a single, monolithic model that relies on optimizations such as multi-threading, this is complicated by the fact that the logical targets of such optimizations—say, repeated instances of processor cores—do not comprise the entire system. Managing the threaded execution of a subset of the design significantly complicates such a model; therefore, it is no surprise that existing threaded simulators have all been hybrid designs, composing a threaded model of the duplicated instances with other elements of a larger host simulator [71, 22, 83].

While these hand-designed simulators provide empirical evidence for the utility of hybrid simulators in enabling optimizations, this dissertation hypothesizes that their advantage is even greater if the optimizations can be applied automatically. The success of peephole optimization [69] in software compilers has shown that separation of concerns and limitation of scope can make optimizing compilers more tractable to implement. By analogue, attempting to thread a set of instances in a simulator compiler would likely be simplified by the ability to rely on composition and therefore disregard the implementation details of the the remainder of the simulator. Therefore, we contend that a compiler capable of directly producing hybrid simulators is a necessary prerequisite to automatic resource optimization of FPGA simulators.

3.2 Compiling Hybrid Simulators with Golden Gate

While automated tools are readily available to convert RTL designs to direct FPGA implementations, prior work on hybrid simulators has generally relegated compilers to a more limited role. Advanced simulators using decoupling to mix heterogeneous models have been carefully designed from the ground up, including numerous examples from the RAMP project [20, 71, 22, 83]. More recently, the MIDAS compiler has automated the generation of a decoupled simulation model for an arbitrary RTL block [54]; while this model may be composed with others in a larger hybrid simulator, MIDAS cannot handle any internal decoupling. With Golden Gate, we provide a compiler framework capable of transforming and optimizing hybrid simulators, allowing RAMP-style resource optimizations to be automatically applied to sub-components of the target design. By introducing internal decoupling to the design, different parts of the target design may be optimized for the host platform in heterogeneous ways. Automating this approach presents two main challenges:

1. Automatically decomposing the simulator into a network of decoupled components. By producing such a network directly from an input RTL specification, the compiler effectively synthesizes a hybrid simulator.

2. Expressing optimizations—such as time-multiplexing—as transforms that modify the components yet still maintain the correct, cycle-exact behavior of the whole simulator.

To ensure that Golden Gate can produce robust, optimized FPGA simulators with no human intervention, we must rely on a formalism that addresses both of these challenges. Specifically, the compiler must rely on a correctness model not only for mapping an RTL specification to a decoupled simulator implementation, but also to govern how it may legally subdivide this simulator into multiple networked components. Therefore, we model our system as a Latency-Insensitive Bounded Dataflow Network (LI-BDN) [89], a type of dataflow network capable of emulating Synchronous State Machines (SSMs) and provide both cycle accuracy and forward progress guarantees. While other formalisms may exist for demonstrating the theoretical soundness of FPGA simulation tools, Golden Gate uses the LI-BDN abstraction to map target designs onto hybrid simulators with high degrees of internal decoupling, enabling heterogenous optimization of different components of the simulator. Indeed, the ability to ease optimization by modularizing simulator implementations was cited as a key motivation in the original presentation of the framework. Furthermore, the well-defined properties of LI-BDNs aid in simulator verification, as discussed in Chapter 9.

3.3 Latency-Insensitive Bounded Dataflow Networks

LI-BDNs are a class of dataflow networks that may be constructed in correspondence to and represent the behavior of arbitrary synchronous circuits. In this capacity, they implement a deadlock-free, cycle-accurate simulation of reference RTL design, while allowing the underlying implementation to use variable-latency handshaking interfaces among its constituent sub-components. Furthermore, LI-BDNs provide a mechanism for arbitrary partitions of the input circuit to be mapped to latency-insensitive implementations, while rules of composition that guarantee that the composite LI-BDN will correctly simulate the full input design in aggregate.

A general LI-BDN is defined by a set of restrictions:

1. Nodes of an LI-BDN are connected via bounded queues.
2. Each node of a LI-BDN must itself be an LI-BDN.
3. The base case is a *primitive LI-BDN*, which is a circuit where all I/O is mediated over handshaking interfaces or *channels*.

In this work, we construct simulators structured as LI-BDNs; therefore, we rely on primitive LI-BDNs to simulate individual partitions of the target design. To relate this to the terminology of the rest of this work, each primitive LI-BDN that simulates a block of hardware is also a *simulation model*. As shown in Figure 3.1, this partitioning of the design yields graph cuts that define the I/O of each partition and its connectivity with other partitions.

Each primitive LI-BDN will have a set of channels that directly correspond with the I/O boundary of the partition it simulates, and the behavior of the primitive LI-BDN will be defined by a functional relationship between the sequence of tokens exchanged across these channels and the values of the corresponding I/Os in the simulated trace.

Partial Implementation

In the presentation of LI-BDNs as a conceptual basis for simulation, this functional relationship is defined as *Partial Implementation* (PI). The original definition [89] is quoted below; here, an SSM S may be an entire target design or a well-defined partition of a larger circuit.

> A BDN R partially implements an SSM S iff
>
> 1. There is a bijective mapping between the inputs of S and R, and a bijective mapping between the outputs of S and R.
> 2. The output histories of S and R matches whenever the input histories match.

More colloquially, partial implementation implies that a latency-insensitive simulation model will produce output tokens sufficient to reconstruct the outputs of a synchronous block exactly as they would appear in a trace where the block received inputs corresponding with the input tokens received by the model. In short, the PI property is a formal way of defining that the model is a cycle-accurate decoupled simulator of the corresponding synchronous block.

While this functional relationship ensures accuracy, it is insufficient to guarantee forward progress. Indeed, since simulators of state machines generally rely on fine-grained synchronization, dividing work across multiple parallel implementations can easily lead to deadlock. While simple solutions include restrictions on the graph cuts dividing work—e.g, dividing the target design only where signals cross a register or queue—the LI-BDN simulation model allows for arbitrary partitioning. Instead, it requires that each primitive LI-BDN obey two more properties: No Extraneous Dependencies and Self Cleaning.

No Extraneous Dependencies

As with the PI property, the No Extraneous Dependencies (NED) property is defined with respect to both an LI-BDN simulation model R and the synchronous state machine S it simulates. This relies on the fact that certain tokens produced by the LI-BDN carry representations (defined by a bijective mapping) of the values of outputs of the SSM on a particular cycle, and that a representation of each input value received by the SSM is carried in some consumed token.

At a high level, the NED property establishes the condition under which an LI-BDN is obligated to produce a valid token carrying the representation of the value of a particular output at a particular point in time within some finite amount of time. This condition depends on the arrival of a particular set of input tokens that is defined by the structure of

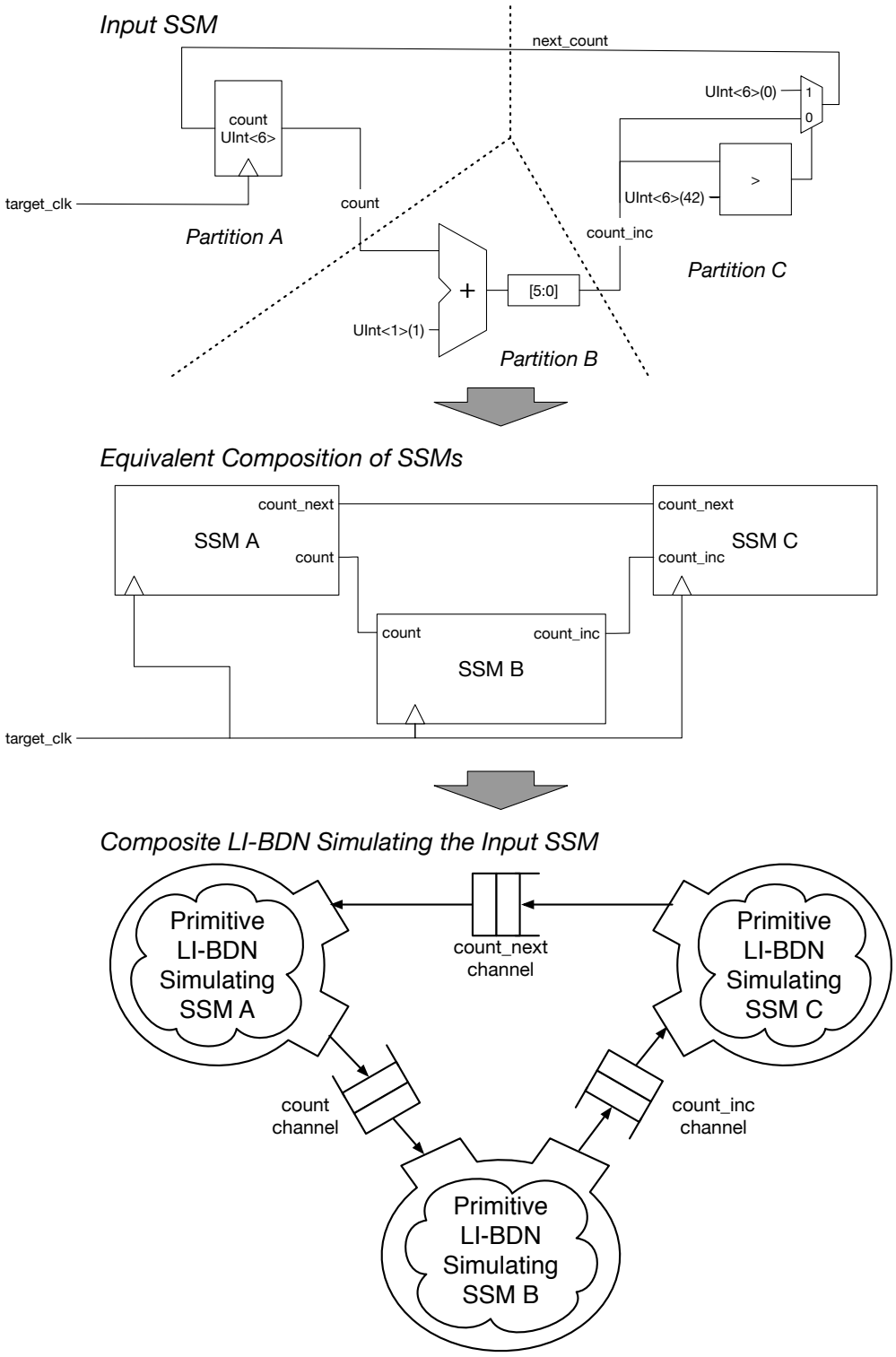

Figure 3.1: A conceptual representation of a simulator for a single SSM—in this case, a 6-bit, free-running counter that wraps when `count == 42`—that is implemented by a composite LI-BDN. The properties that each primitive LI-BDN is guaranteed to obey allow this composition to deterministically simulate the original SSM without risk of deadlock.

the SSM: specifically, the set of all tokens carrying the representations of the inputs that are combinationally connected to the output in question.

This definition [89] of this set for a particular output O_i is quoted below:

> For any output O_i of a primitive BDN R which implements SSM S, `CombinationallyConnected`(O_i) is the inputs of R corresponding to those inputs of S that are combinationally connected to the output O_i in S.

With this definition for `CombinationallyConnected`(O_i), the definition of the NED property reflects the fact that if the LI-BDN model has successfully simulated cycle N, it is possible to produce a valid token representing the value of O_i at cycle $N + 1$ as soon as all the tokens representing the values at cycle $N + 1$ of all the inputs combinationally connected to O_i. This can in turn be translated to a property that deals only with tokens.

Definition of No Extraneous Dependencies: if R has produced at least N tokens on each output channel, it is possible to produce a valid token representing the value of O_i at cycle $N + 1$ as soon as all the tokens representing the values at cycle $N + 1$ of all the inputs combinationally connected to O_i.

If the model fails to produce this token within some finite latency after this condition is satisfied, it has *extraneous dependencies*; since the deadlock-avoidance guarantees of LI-BDNs do not hold if such dependencies exist, they are disallowed by the NED property. This requirement is a key point of contrast with A-Port networks; while both systems define mechanisms for distributing simulator control, A-Port models are allowed to wait until all inputs for a given time step are ready before producing any outputs. Since this could potentially result in circular dependencies if there were any combination connections between input and outputs, A-Port networks restrict legal partitions in the simulator. In contrast, LI-BDNs allow for arbitrary cuts, which can ease certain optimizations, including the modeling of FPGA-hostile combinational-read memories presented in Chapter 5.

Self Cleaning

While the NED property defines a condition under which the LI-BDN model is obligated to produce a token representing the value of a particular output, the Self-Cleaning (SC) property outlines when it is obligated to consume a token that has arrived. By limiting the circumstances under which an LI-BDN is allowed to exert back-pressure to upstream producers, it prevents certain classes of deadlock-causing circular dependencies.

Definition of Self Cleaning: Assume that each input channel of R is connected to an unbounded stream of input tokens. In this case, if R has produced at least N tokens on each output channel, it is obligated to consume at least N tokens on each input latency within some finite latency.

If a simulation model that partially implements an SSM S is a true primitive LI-BDN—i.e., if it obeys both the NED and SC properties—it *fully implements* S. Furthermore, the rules of composition imply that two LI-BDNs that each implement an SSM can implement a circuit composed of the two SSMs if the connectivitivity of the channels corresponds directly with the connectivity of the I/Os of the two SSMs. As shown in Figure 3.1, this composition allows the simulation of a circuit to be divided over an arbitrary number of primitive LI-BDNs.

3.3.1 Combining Multiple Signals Within Channels

In the formal justification for LI-BDN simulation, the proof that the NED and SC properties are sufficient to avoid deadlock depends on the fact that a valid SSM may not contain combinational loops. In particular, the guarantee that the NED property is sufficient to avoid deadlock depends on the fact that O_i may not be part of the transitive closure of `CombinationallyConnected`(O_i); a violation of this requirement would both imply the existence of a combinational loop and leave open the possiblity of a circular dependency in the decoupled simulator.

While bitvector (multi-bit) signals or interfaces represent a convenient abstraction for RTL designers, this presence or absence of combinational loops is defined in terms of individual two-state binary signals, which is consistent with the fact that combinational loop detection is often performed on circuit netlists. This in turn implies that each output O_i and each element of its associated set `CombinationallyConnected`(O_i) is defined within the NED property at the bit level. This has an unfortunate consequence: though it would be more resource-effecient to represent each multi-bit I/O in the RTL specification of the SSM with a single channel, combining these formerly independent signals can lead to extraneous dependencies. As shown in Figure 3.2, a circuit that lacks true combinational loops may result in cycles when translated to a graph representing multi-bit signals and their combinational dependencies.

Fortunately, the graph representation shown in Figure 3.2 also yields a solution to this issue. Since the proof that the NED property prevents deadlock due to circular waits for valid tokens does not depend on the binary nature of the individual signals, combining the multiple bits of a bitvector I/O into a single channel only poses issues when the bitvector dependency graph itself contains circular dependencies. While the use of standard combinational loop detection is insufficent to guarantee freedom from such bitvector dependency loops, it is possible to materialize the associated graph representation and find its strongly connected components, and the lack of resulting cycles allows the NED property to be defined in terms of channels representing bitvector signals. While this check would reject the circuit in Figure 3.2, signals found along cycles may be broken into constituent single-bit nets to restore an acyclic dependency graph.

Fortunately, this bitvector-level combinational dependency loop checking is included within the standard FIRRTL distribution, allowing Golden Gate to analyze circuits and ascertain whether it is legal to generate LI-BDN models that avoid splitting bitvector signals

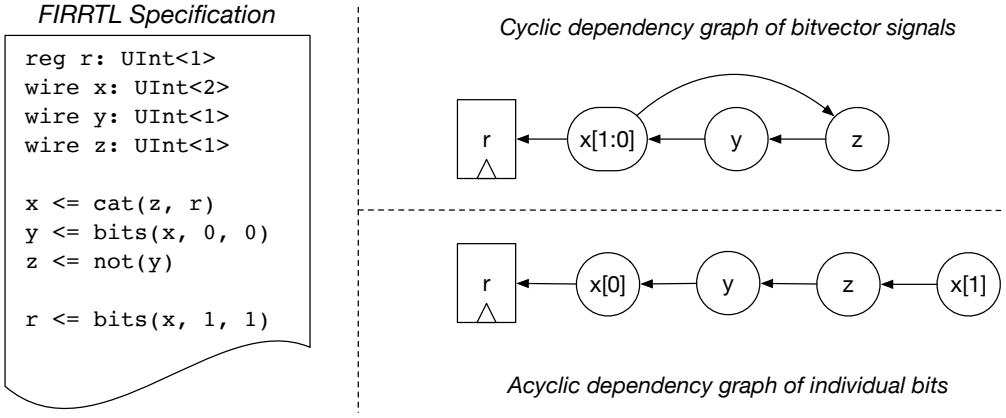

Figure 3.2: An illustration of a circuit that is free of combinational loops but results in cycles in a graph of the combinational dependencies of each named bitvector signal. While the existence of higher-level composite types such as vectors renders these patterns less common in FIRRTL than in Verilog, the occurrence of such cycles makes it impossible to represent `x` with a single channel.

into multiple channels. Furthermore, it is possible to coalesce any set of inputs or any set of outputs on a partition into a single node; if the resulting graph is acyclic, it is legal to pack the representation of the full set within a single channel. These techniques allow the compiler to minimize resource overhead by reducing the number of channels and associated queues in the LI-BDN simulator.

3.3.2 Multiple Clock Domains

Throughout the previous sections, LI-BDNs have been presented as a tool to simulate synchronous state machines with networks of decoupled models. However, a modern SoC is essentially never a monolithic synchronous design; multiple clock domains are a common feature, and components such as Phase-Locked Loops (PLLs), clock multiplexers, and asynchronously reset register are ubiquitous. While the prevalence of Globally Asynchronous, Locally Asynchronous (GALS) design principles [14] can allow a purely synchronous simulation tool to simulate large components of a system, a critical goal of FireSim and Golden Gate is to model full, realistic target SoCs.

Fortunately, related work on Golden Gate involves extending it to support modeling these difficult features. By incorporating a variation of the Chandy-Misra-Bryant simulation protocol [17, 10], a technique that relies on explicitly timestamped messages to synchronize parallel (often software) implementations of discrete event simulations, Golden Gate may be extended to model multiple clock domains by introducing a notion of absolute time to the simulation. Furthermore, truly asynchronous devices like PLLs and clock multiplexers can be modeled as being synchronous to a virtual, high-frequency clock; while this reduces

fidelity to the physical chip, it is consistent with the use of finite timescales in software RTL simulation and provides the benefit of deterministic simulation. While these additional features complicate the implementation of both the simulation models and associated channels, this extension also defines protocols for crossing between a pure LI-BDN modeling a locally synchronous block of the system and the asynchronous global simulation regime, allowing the optimizations presented in this dissertation to be incorporated within individual domains of a GALS target design.

3.4 Compiling Target RTL to LI-BDN Simulators

While Vijayaraghavan et al. [89] introduce the concept of emulating synchronous circuits with LI-BDNs, they stop short of implementing that concept in a real simulator; others have since done so in handwritten simulators. To the best of our knowledge, Golden Gate is the first tool to *automatically* produce FPGA-accelerated simulators that are structured as LI-BDN networks, and additionally, lime is the first tool to formally verify primitive LI-BDNs.

Given the focus on deadlock-free, cycle-accurate simulation of SSMs, LI-BDN simulators naturally occupy the "concrete RTL" portion of the simulator design space. As described in more detail in Chapter 4, Golden Gate compiles concrete target RTL inputs to LI-BDNs by dividing the circuit into several SSM partitions, mapping each partition to a primitive LI-BDN, and composing these primitive LI-BDNs into a larger network that simulates the entire input RTL circuit. By adhering to the properties discussed above, components of this modular simulator may be individually refined with heterogeneous optimizations to produce a resource-efficient hybrid simulator.

Chapter 4

The Golden Gate Toolchain

A major barrier to FPGA prototyping is the necessity to buy in to a proprietary host FPGA or private cloud platform. Therefore, Golden Gate is implemented as an extension to FireSim [49], an open-source tool that enables system designers to simulate their target RTL designs on commodity FPGAs hosted in Amazon's AWS public cloud. While FireSim already relies on the decoupled simulation paradigm to support co-simulation of models of network and DRAM interfaces, it does not actually apply any optimizations to the transformed target design, so the resource utilization of the target RTL is nearly identical to an FPGA prototype. In contrast, Golden Gate adds an optimizing compiler to significantly reduce resource utilization with minimal engineering effort.

In order to make Golden Gate as widely applicable as possible, it is designed to support a variety of optimizations across different FPGA host platforms. The extensible nature of the compiler makes it easier to add new optimizations, which are intriniscally supported by the push-button lime flow.

4.1 FireSim and Golden Gate

As discussed in Section 2.1.5, FireSim is an open-source FPGA simulation platform for modeling modern SoCs that relies on FPGAs hosted in the Amazon Web Services (AWS) public cloud. FireSim provides cycle-accurate simulation of systems defined by full RTL implementations, but it also defines a deterministic, cycle-level interface for integrating co-simulated, mixed hardware-software models—or *bridges*—of components that are prohibitive to implement in RTL.

In order to provide a useful starting point for academic research or commercial use, FireSim aims to provide not only a simulation platform, but a suite of target designs, target software, and related tools. When combined with both the suite of CAD tools required to produce a functioning FireSim simulator, the library of supporting components to manage integrating the simulator with the host AWS F1 cloud computer instance, implementations of various bridges, and a vast collection of scripts to automate the process of building and

running simulations for the end user, it is clear that the full FireSim stack consists of many layers.

As shown in Figure 4.1, this complexity is reflected in the process of transforming a target design into a concrete FireSim simulator, which incorporates several different tools and stages. In this work, we focus on Golden Gate, the compiler responsible for translating the major portion of the target system that is defined in RTL into a hardware implementation of a decoupled FPGA simulator. Given the emphasis on cycle-accurate, RTL-based simulation, this concretely specified partition represents the majority of the functionality of the target system; furthermore, it accounts for nearly the entire FPGA resource footprint of the simulator. Therefore, with the ultimate goal of allowing FireSim to simulate larger systems on finite-sized FPGA devices, we explore Golden Gate as a platform for reducing the resource footprint of the FPGA implementation that simulates the RTL specification of the majority of the target system.

4.2 A FIRRTL-Based Simulator Compiler

As discussed in the previous section, the definition of the simulation environment is complicated by the desire to co-simulate components of the target design across both software and FPGA host platforms. While the overall FireSim system can be viewed as a "closed world" simulator of a full target system, Golden Gate produces a hardware implementation of a simulator of the portion of the target design that is specified via the set of components described in the input RTL. This simulator interacts with an environment that models the rest of the target design. Though the implementation of this environment is outside the scope of Golden Gate and includes both specialized hardware and software models, well-defined interfaces allow these components to effectively compose to form a simulator of the broader system.

Within the broad scope of the FireSim FPGA simulation platform, Golden Gate performs the task of transforming an input RTL *target design* into a concrete implementation of an FPGA simulator of that RTL design. Throughout this work, we rely upon FIRRTL (see Section 2.3) as a convenient infrastructure for expressing compiler transformations on digital circuits; therefore, the input design and output simulator implementation are both FIRRTL circuits. While many components of FireSim, such as target design generators, co-simulation models, and software are specialized to particular target designs, Golden Gate is capable of producing optimized, decoupled FPGA simulators from arbitrary FIRRTL inputs; in light of this generality, we refer to it as a *simulator compiler*. Since Golden Gate is implemented as a collection of FIRRTL transforms, it relies on several key features of the associated software infrastructure, including fine-grained incremental lowering, an extensible annotation interface, anda collection of built-in analyses and consistency checks.

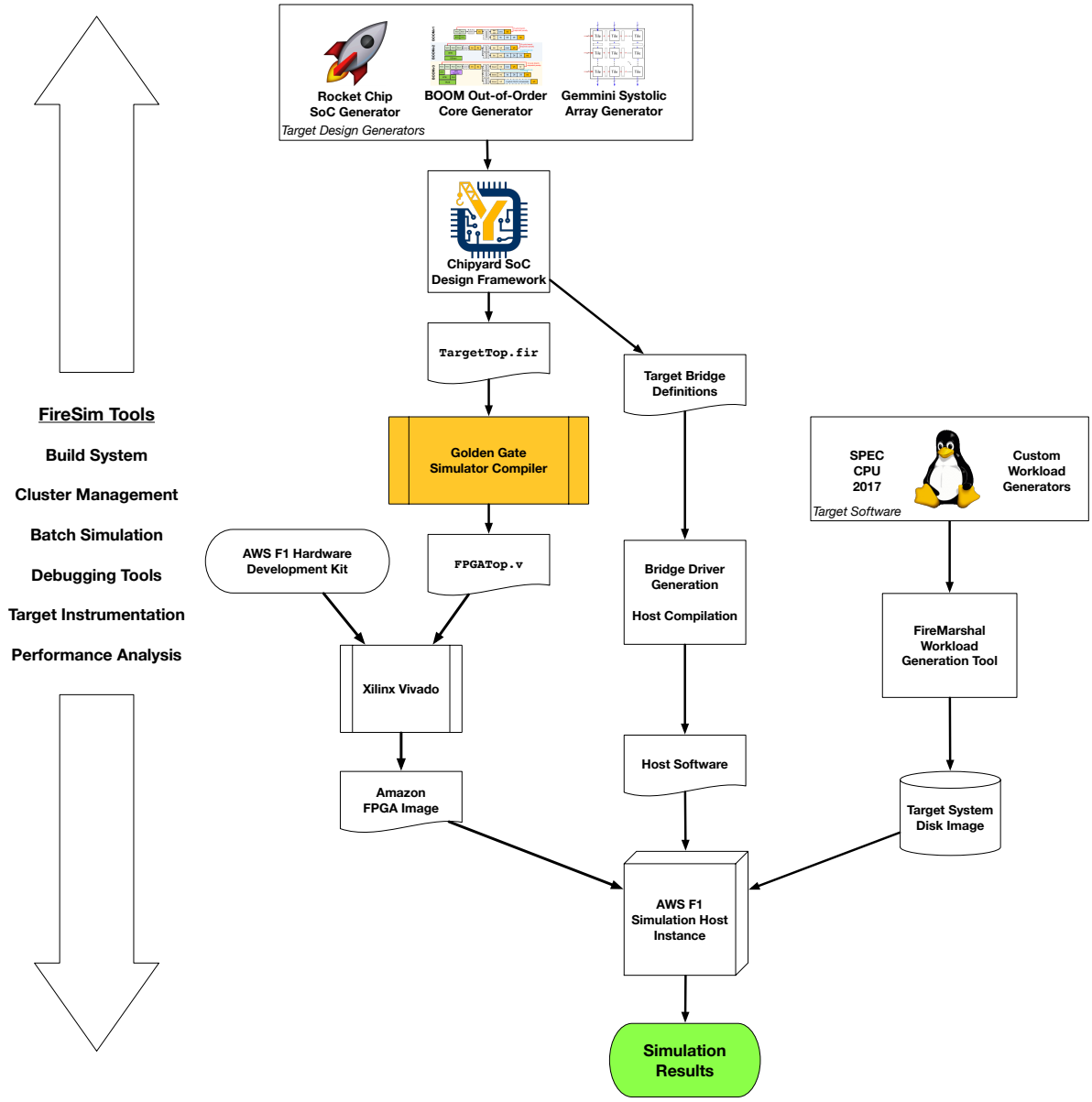

Figure 4.1: A high-level overview of the full FireSim stack is used to generate a complete simulation run. This work focuses on enhancing the Golden Gate simulator compiler with resource-saving optimizations that increase the simulation capacity of the FPGA host platform.

4.2.1 Fine-Grained Incremental Lowering with Core FIRRTL Passes

While FIRRTL IR is simpler than most HDLs, it includes a number of higher-level features, such as aggregate types (vectors and bundles of signals) and multiple assignment of signals. In order to simplify the implementation of a given source-to-source transformation, it is often desirable to make several simplifying assumptions about the circuit that limit the space of language semantics it must consider. Beyond the presence or absence of specific language features, such assumptions can have more nuanced definitions, such as assuming the lack of the use of a particular operator on a particular type of signal.

In order to address the need of FIRRTL-based tools to express such simplifying assumptions, the reference FIRRTL compiler is build around a library of fine-grained *lowering passes*, which incrementally simplify the circuit by removing features or ensuring that simplifying properties—such as guaranteeing single static assignment [75] (SSA) of components—hold across some portion of the circuit. In many cases, such simplifications are mutually orthogonal; however, some carry transitive dependencies: in the example of converting to SSA form, it is necessary to first simplify "bulk connections" that act upon bi-directional interfaces. In order to allow transforms to make such simplifying assumptions, the FIRRTL software infrastructure requires it to declare its *prerequisites*: the set of transforms that must be applied to the circuit before the transform in question.

While the notion of prerequisites allows a transform to make simplifying assumptions about appropriately lowered circuits, a recurring pattern is where a transform has a particular simplifying assumption as a precondition that is not held as a postcondition. For example, several transforms in Golden Gate require that aggregate-typed ports on modules be lowered to more basic types, yet some of those transforms generate new ports with aggregate types. This pattern of non-monotonic lowering is well supported in FIRRTL [47], and it is encoded in the software infrastructure through the declaration of an *invalidation function*, which may declare that a transform invalidates one or more of its prerequisites or any other transforms.

In contrast with the prerequisites of a transform, which are declared as a static list, the invalidation function is a dynamic function whose abstract interface is to accept a transform and return a Boolean indicating whether the current transform invalidates it. This has several advantages over a static list in a flow that integrates multiple custom transforms, since any transform that modifies the circuit cannot be aware of every possible transform that it invalidates; this uncertainty can be resolved with type casing, where a transform declares that it invalidates certain classes of transforms or even all transforms outside of certain classes, such as the built-in transforms of the reference compiler.

Taken together, these features of the FIRRTL software infrastructure enable and encourage the development of fine-grained transforms. Though the FIRRTL specification defines only three official "forms" corresponding with different lowering processes (High, Middle, and Low FIRRTL), there are currently 27 transforms in the core compiler that incrementally lower the circuit by removing features and simplifying interactions of features—this

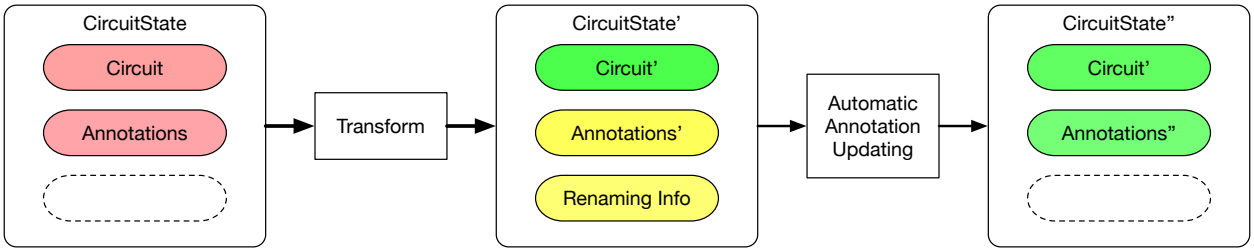

Figure 4.2: Structure of a FIRRTL transform: the transform accepts an in-memory representation of a circuit in FIRRTL IR that is augmented with core analyses (such as expression tree type) alongside a list of annotations. It emits both a new circuit and a new annotation list that are each the result of applying a pure function to its full input.

number excludes analyses, checks, and optional features. Golden Gate is implemented as a collection of 21 transforms, each of which is responsible for a particular phase of the compilation process; in practice, this modularity of the software implementation helps enforce the modularity of the simulator implementation that is key to composing multiple optimizations.

While granular passes offer many convenient properties, they are not without cost. When provided with a desired list of transformations, a FIRRTL *transform manager* solves a graph problem to pick a valid transform ordering. Though it will attempt to optimize the number of transforms, this ordering will inevitably involve repetition of transforms due to the repeated re-lowering patterns discussed in this section. While the full Golden Gate flow invokes only 97 distinct transforms, the total dynamic count of all transform iterations is much larger at 571 transform executions—predominantly due to repeated invocations of built-in FIRRTL lowering, analysis, and check passes. However, this cost is not significant to the total FPGA simulation design loop; while the Golden Gate runtime is measured in minutes, the runtime for downstream FPGA tools is measured in tens of hours.

4.2.2 Harnessing the Extensible FIRRTL Annotation Interface

Since the FIRRTL infrastructure encourages dividing a custom flow into a number of modular transforms, each with a limited concerns, it naturally follows that there will be a larger number of points where one transform communicates information to another. While this communication centers around each transform providing its FIRRTL IR output as the input to the succeeding transform, this can be excessively limiting. A given transform may produce data structures that need to be shared with another, or a flow may require the parameterization of the behavior of transform, such as limiting the scope of lowering to a particular module. Given the goal of producing a flexible infrastructure, it is impractical to rigidly codify all such communication in the formal IR. Instead, FIRRTL relies on *annotations* to communicate information alongside the IR.

At a high level, annotations are user-defined channels of information of nearly arbitrary structure between transforms. When they refer to components in the circuit, they do so

through indirect references known as *targets*. While the notion of a target is the only standardized requirement for the data contained in an annotation, it imposes a key requirement: that the compiler can enumerate all hardware components that are associated with a given annotation. As shown in Figure 4.2, a transform receives a list of annotations alongside the IR of the input circuit and outputs a new list (which may be the same) alongside the transformed IR. However, since this effectively presents an API that must be respected, their use in the compiler is standardized and subject to several constraints.

In general, a transform concerns itself only with the limited set of annotations that form part of its defined API. For the example of limiting the scope of a lowering transform to a particular module, that transform may declare a class of annotation that allows a user or upstream transform to label the set of modules upon which it should operate. It is important to note that this does come at the cost of significantly expanding the interface of a given transform to include all of its associated annotations: in order to productively use it, it is no longer always sufficient to simply run the transform. As a result, it is advantageous to limit the number of classes of annotations that may affect the behavior of a particular transform.

Since the set of annotation classes is subject to arbitrary extension, a transform will receive inputs for which it has no semantic understanding. In general, these extraneous annotations will be passed through to the output. However, a transform that modifies the circuit may create an obligation for an unassociated annotation to be updated to remain consistent with the circuit, which presents an implementation challenge.

In order to accommodate these updates, FIRRTL provides several conventions for annotation management [46]. While this annotation management infrastructure is an area of active development outside the scope of this work, there are several useful patterns. First, a transform that modifies the circuit in a widely seen way, such as renaming or removing a component, can rely on a built-in *renaming* API. This interface lets a transform declare the list of such modifications it applied to the circuit, and all annotations that refer to the modified component via targets will be appropriately updated via a set of pre-defined rules. Furthermore, a transform that undoes the work of another transform does not need to enumerate its annotations. With the exception of renames, a transform need only directly manage the consistency of the annotations it produces; a transform that modifies the circuit should invalidate other transforms that generate annotations encoding the structure of the circuit.

In both cases, the utility of the convention depends on a taxonomy of targets, annotations, and transforms, as a lack of structure would lead to all transforms invalidating all other transforms. This taxonomy, which includes notions such as analysis transforms, analysis annotations, core compiler transforms, and monotonic lowering transforms is an area of ongoing research. However, the use of Scala as a host language makes the maintenance of such a system more straightforward than in a traditional object-oriented language with features like mixin traits, pattern matching, and a rich type system.

4.2.3 Built-in FIRRTL Analyses and Consistency Checks

Though the extensible FIRRTL infrastructure enables a complex flow to easily be divided into sequenced transformations, many recurring patterns appear in their implementations. Since many transforms often wish to perform the same type of analysis about the static or structural properties of a circuit, the most common are made available as reusable transforms, including finding combinational paths within a module and elaborating instance hierarchies in a design.

Each *analysis transform* has three properties: it does not modify the circuit, it stores the result of its analysis in a structured annotation, and it replaces any such existing annotation with an up-to-date version. Furthermore, any transform that modifies the circuit in such a way as to change the results of such an analysis must declare that it invalidates the analysis. Rather than enforcing this via an enumeration of all possible analyses, a transform that modifies the structure of the circuit can use pattern matching to invalidate all analyses other than whichever it is known to preserve; while this idiom—directly inspired by the management of analysis passes in LLVM [59]—carries a runtime cost, these transform-based analyses are generally utilized across short phases of the compiler and represent a small fraction of overall compiler runtime. Given these properties, a transform that requires the results of a particular analysis can simply declare it as a prerequisite, recover the analysis from the circuit's annotations, and immediately utilize the associate data structures.

While many analyses are suited to the model of invalidation and re-computation for use in a limited phase, some analyses are so fundamental that the burden of frequently re-analyzing the entire circuit would significantly slow the compilation process. For example, both core and custom transforms often desire to associate information about a declared component (e.g., a register or wire) with expressions that operate on or modify the value of that component. Therefore, the FIRRTL compiler also promotes a fixed set of analyses to first-class properties of the in-memory representation of any expression tree or subtree: its data type, its usage as either the driving or driven signal in the enclosing statement context, and, in the case of a variable reference, the kind of declared component (e.g., port, wire, register) it is associated with. Since these analyses are fixed in number, a transform that modifies the circuit may update them in the "peephole" view of the circuit that it modifies, rather than invalidating them entirely.

In practice, Golden Gate relies heavily on this library of built-in analyses. The type information and drive directions carried in the in-memory IR are used in nearly every single custom transform, while both the instance hierarchy and combinational path analyses mentioned above are critical information for the aspects of Golden Gate that modify the hierarchy of the circuit. Finally, FIRRTL exposes many analyses via the set of built-in check transforms, which help identify points of failure in custom compiler flows.

4.2.4 Differing Requirements of RTL Compilers and Simulator Compilers

Unlike most previous work in FIRRTL-based flows, which produce a logically equivalent output circuit after applying some set of lowering transformations, the output of Golden Gate is a FIRRTL implementation of a *simulator*. Though FIRRTL is explicitly aimed at producing custom compilation flows for RTL circuits, the process of producing an optimized FPGA simulator represents a fundamentally new use case for FIRRTL, with the key difference depicted in Figure 4.3. More specifically, while a simulator compiler is bound by the same "as-if" behavioral guarantees that constrain all optimizing compilers [45], a typical FIRRTL *RTL compiler* requires not only that its output be logically equivalent at the register-transfer level, but that there is a perfect cycle-by-cycle state correspondence between the input and output circuits. While more aggressive RTL compilers can apply optimizations such as register retiming [64] or substitution of individual components that relax this requirement, the fundamental guarantee is that the output circuit will not produce any observable differences from interactions with the environment through an interface that is defined by register-transfer level semantics. For the synchronous digital FIRRTL circuits, traces of interactions with the environment are defined at a cycle-by-cycle level; in other words, equivalence of input and output circuits is defined strictly at a synchronous level.

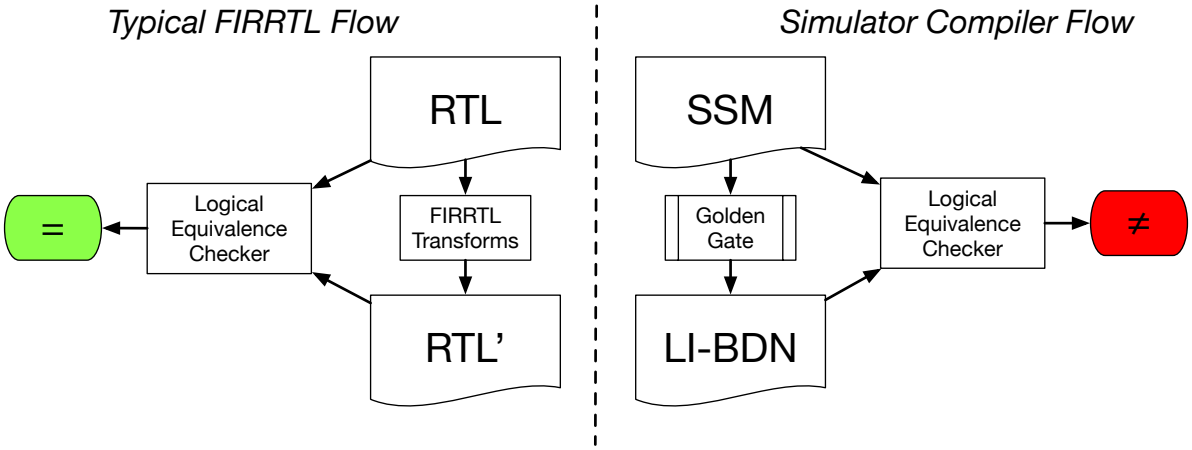

Figure 4.3: A comparison of a typical FIRRTL flow with a simulator compiler. Unlike a standard FIRRTL transform, where the output is either logically equivalent to the input or represents a refinement of its RTL specification, the output of a simulator compiler is a specialized hardware implementation that is capable of simulating the input RTL in a cycle-accurate manner. In this case, the output LI-BDN implements the input SSM, but its concrete implementation has a distinct RTL specification.

In contrast, a simulator compiler is free to rely on a different model: the interactions of the simulator implementation with the simulation environment merely need to encode this precise, synchronous behavior of the target design. In the context of a producing a hardware

implementation of a simulator specific to a particular input circuit, this implies that the generated circuit must accept inputs representing the inputs of the circuit and produce outputs representing the outputs of the circuit, and it is free to encode these representations in any synchronous or asynchronous protocol defined by the simulation environment.

Putting this abstract notion in more concrete terms, while most FIRRTL-based compilers will transform input circuits into output circuits that can be defined as equivalent via FIRRTL semantics alone, Golden Gate will transform the input circuit into a simulator that produces equivalent behavior in the context of an higher-level interface with its environment. As discussed in Chapter 3, we rely on the notion of an LI-BDN implementation of the target design to define this interface. Therefore, the output of Golden Gate is a hardware implementation of an LI-BDN that *implements* the target design per the definition in Section 3.3.

While this is a departure from the behavior of most FIRRTL transformations, it highlights a key advantage of FIRRTL over netlist formats: the ability to rely on *source-to-source* transformations rather than structural circuit transformations. Though LI-BDN implementations of a circuit could take on infinite forms, the goal of producing a reusable compiler requires a flow that can produce one via successive transformations of the input circuit. Since Golden Gate is a FIRRTL transpiler, large blocks of FIRRTL code may pass through unchanged in order to do the "heavy lifting" of implementing much of the logic of the target design. While small changes in behavior will often require complex traversals of a netlist, FIRRTL idioms like syntactic pattern matching and insertion of higher-level constructions are very useful for making radical changes to specific portions of a design.

4.3 Compiler Organization

A key constraint of the Golden Gate toolchain is that each model in the simulator will be responsible for simulating a partition of the target design. This structural approach provides separation of concerns, simplifying the task of generating each (potentially optimized) simulation model. However, since the compiler is structured around FIRRTL source-to-source transformations that act on modules, rather than netlists, the hierarchy of the target design can limit the possible topologies of the simulator, and therefore the ability to apply optimizations at a fine granularity.

Fortunately, FIRRTL transforms provide the ability to modify the hierarchy of the target design. Furthermore, it is possible to perform these mutations as a pre-processing step before any simulation models are generated. While this separation of concerns is in keeping with the general FIRRTL design philosophy of creating layered compiler flows, it also allows the compiler to re-structure the hierarchy and partition the design while maintaining its logical equivalence with the original specification.

With this in mind, the Golden Gate compiler is divided into two main phases: *Target Transformation* and *Simulator Synthesis*, as shown in Figure 4.4. The Target Transformation phase is structured as a typical FIRRTL flow, where custom modifications to the target design

still preserve RTL semantics. The passes that make up this phase perform much of the heavy lifting in restructuring and partitioning the circuit, and they can be thoroughly tested by running off-the-shelf Logical Equivalence Checking (LEC) tools on input-output circuit pairs. This reduces the footprint and development risk of the code in the later Simulator Synthesis phases, where the compiler translates the target to a FIRRTL design that is not logically equivalent under FIRRTL RTL semantics but instead implements a simulator capable of precisely modeling its RTL behavior.

4.3.1 Target Transformation

While the Golden Gate flow allows arbitrary components of the target design to be marked for promotion, causing them to each be simulated by a dedicated simulation model, division of the simulator into multiple models is generally driven by the labeling of blocks as targets for optimization or the presence of quasi-RTL features such as multiple clock domains. In this work, we focus on resource-conserving optimizations; therefore, the partitioning of the design is structured around moving each optimization target into an independent "optimization domain." Therefore, during the Target Transformation phase, optimization candidates are identified and the target's module hierarchy is mutated into a structure that ultimately reflects the topology of the final composite LI-BDN.

Target transformations are performed as a series of small operations that consume and generate metadata encoded in FIRRTL annotation. In order to identify optimization targets, each optimization has a corresponding analysis pass, which inspects the circuit and consumes designer-provided hints captured—expressed via annotations in the Chisel circuit—or pattern-matches blocks of hardware that will yield significant resource savings when transformed via a particular optimization strategy. For example, a set of repeated instances of processor cores in an SoC might be labeled with an `EnableModelMultiThreadingAnnotation`, which designates them as targets for the multi-threading optimization discussed in Chapter 6. When the pass finds a candidate sub-circuit, it wraps it in a module and then labels the module with annotations that indicate how it should optimized and how its I/O will correspond to token ports. Once all candidates are identified, the wrapper modules are "promoted" to the top of the module hierarchy. This process is then repeated for the next optimization.

4.3.2 Decomposed Target Form

At the end of target transformation, the RTL is in *decomposed target form*, which is a canonicalization of the target design that mirrors the topology of the desired LI-BDN simulator while still maintaining logical equivalence with the original input. In this form, the main module at the top of the transformed target hierarchy may contain an arbitrary number of instances. By definition, each of these direct child instances of the target will be simulated with a single, dedicated primitive LI-BDN. All modules are labeled with annotations that indicate how they should be transformed and/or optimized to generate this LI-BDN, and

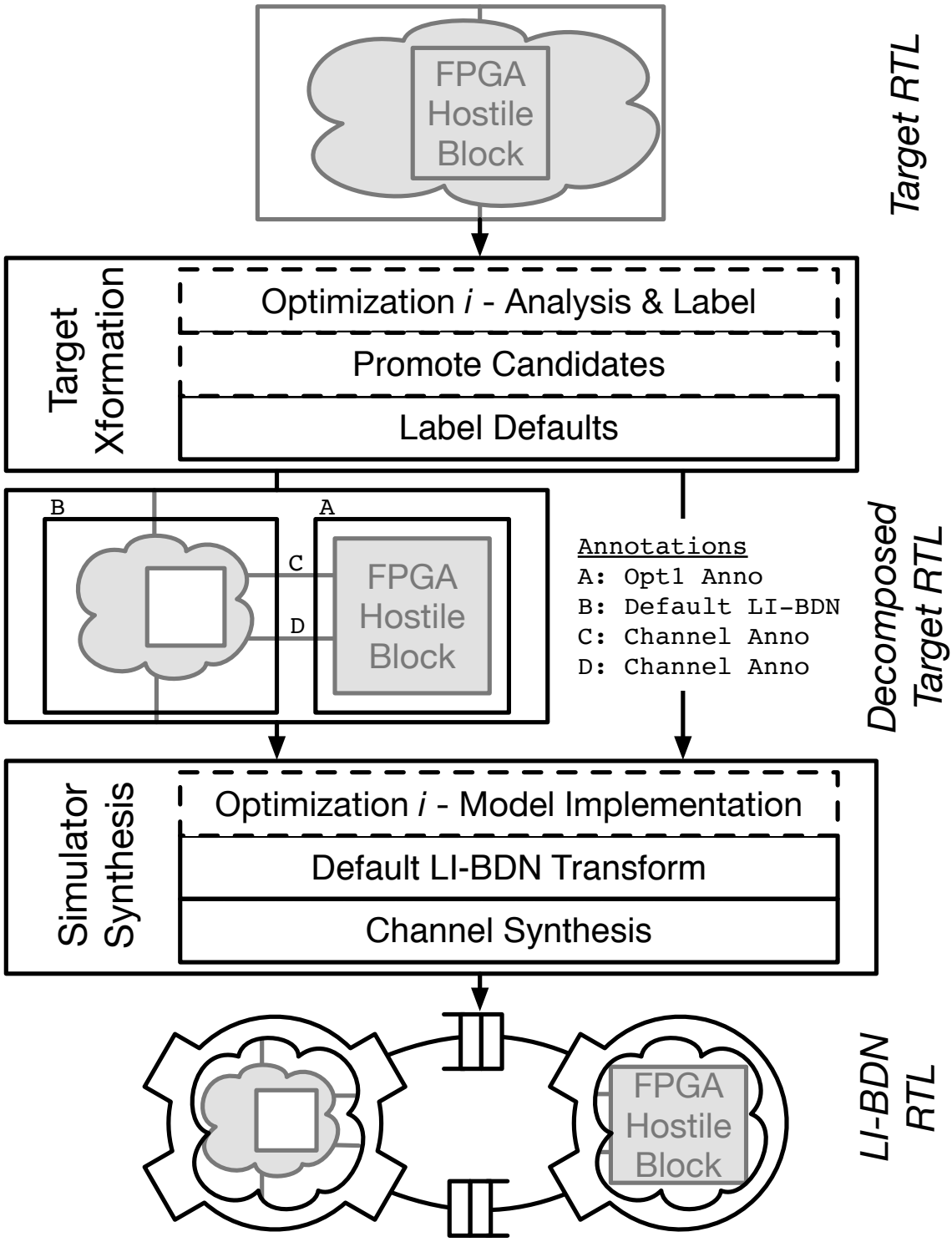

Figure 4.4: The Golden Gate compiler flow

how their inputs and outputs should be coalesced into token channels. Furthermore, the connectivity graph of the I/O different child instances defines the channel structure of for the composite LI-BDN generated during simulator synthesis.

4.3.3 Simulator Synthesis

Here, model-implementation passes construct the LI-BDN by replacing modules with a model based on its annotation. This fundamentally changes the structure of the circuit, but as we will show, this can be verified using LIME (Chapter 9). Model-implementation passes come in two varieties:

1. Transformation-based: these rely on source-to-source FIRRTL transformations. In the simplest case, this would involve augmenting the microarchitecture of the SSM specified by the input RTL for latency insensitivity, but such a transform might also further transform a primitive LI-BDN into an optimized version.

2. Generator-based: these inspect the structure of the target RTL to parameterize an LI-BDN model generator. The output of this generator—which may be written in Chisel—is a complete primitive LI-BDN. This technique would only be used when the resulting model is very distinct microarchitecturally from the original target SSM, as is the case for the multi-ported memory optimization discussed in Chapter 5.

4.4 The Default LI-BDN Transform

The default model-implementation pass is transformation-based and converts a target module into a primitive LI-BDN as follows:

1. For each output channel, it finds all input channels to which it is combinationally connected (CC).

2. For each output channel, it generates a predicate, `firing`, that is asserted when all CC input tokens are available, and a register, `fired`, that is set when that output channel has enqueued but the rest of the model has not yet advanced.

3. It gates all state updates with a `finishing` predicate. When this signal is high, all `fired` bits are reset.

4. It drives `finishing` by taking the conjunction across all output channels of the term $\texttt{fired}_o \vee \texttt{firing}_o$.

4.5 Adding New Optimizations

Ultimately, the goal of Golden Gate is to provide a framework that makes enhancements and modifications relatively tractable. By layering modifications to the target design and iterative refinements of the simulator LI-BDN, optimizations can integrate with the baseline flow without any fundamental code changes. As shown in Figure 4.5, adding a new optimization to Golden Gate requires a few different components to be defined, each interacting with multiple layers of the compiler.

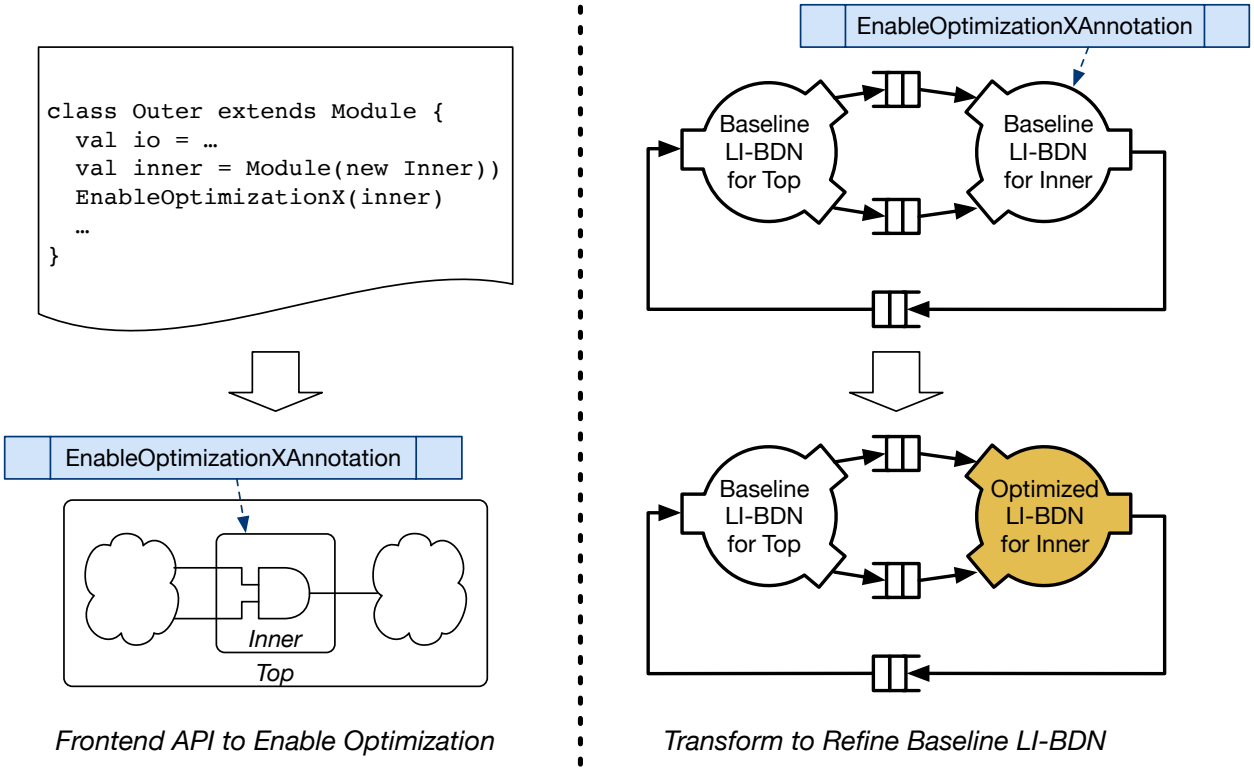

Figure 4.5: The two key components of a Golden Gate optimization: a user-facing API to enable the optimization, and a FIRRTL transform that refines an unoptimized LI-BDN to a resource-conserving, optimized version. Note that the extraction of the target components labeled for optimization and the conversion of the partitioned circuit into a baseline composite LI-BDN is handled by the core compiler flow illustrated in Figure 4.4.

In order to apply any optimization, it is first necessary to identify the partition or partitions of the target design that will each be simulated via an optimized primitive LI-BDN. Given the complex design space resulting from this selection problem, and given the focus on the underlying implementation and optimal performance benefits of each optimization, this work relies on direct annotation of target blocks to denote these partitions. While it is possible for a compiler to apply heuristics to automate this process, this is reserved for

future work. Therefore, the first step in the flow depicted in Figure 4.5 is the generation of a FIRRTL annotation that labels a particular module as a target for optimization. This process relies on a frontend Chisel API that must be written for each optimization; fortunately, this interface may rely on Scala type safety to ensure that the annotations are only applied to components that can be appropriately handled by downstream transformations. As shown in Figure 4.6, various Scala techniques may be used to separate the calls to this API that generate the associated FIRRTL annotations from the Chisel implementation of the target design, eliminating the need to directly modify the RTL for simulation purposes.

```
def enable_simulator_optimizations(target: TargetSystemTopModule) = {
  system.tiles.foreach {
    t =>
      annotate(EnableModelMultiThreadingAnnotation(t))
      annotate(EnableMemoryOptimizationAnnotation(t.core.regfile))
      annotate(EnableMemoryOptimizationAnnotation(t.core.fpRegfile))
  }
}
```

Figure 4.6: A method that can enable the application of Golden Gate simulator optimizations to a target design that has already been elaborated. By employing type-safe cross-module references, this avoids polluting the target RTL with simulation-specific code.

At this point, the annotated target design is passed to the Golden Gate compiler. In order to take advantage of the modularity offered by the LI-BDN simulator model, any hierarchical component of the circuit that is a target for optimization will be simulated by a dedicated primitive LI-BDN. Furthermore, as discussed in Section 4.3.1, the initial Target Transformation phase of the compiler mutates the RTL circuit to produce a design hierarchy that reflects the desired topology of the final composite LI-BDN. Therefore, as shown in Figure 4.5, the labelled component must be extracted to the top level. In order to allow new optimizations to integrate with the compiler and signal the need to perform this transformation, their associated target-level FIRRTL annotations can rely on extending an abstract supertype that generally demarcates the need to simulate a block with its own dedicated primitive LI-BDN.

At the end of the Target Transformation phase, with the block in question now separated at the top level of the hierarchy, the Simulator Synthesis phases begins by transforming each top-level block into a baseline primitive LI-BDN. At this point, the design has been successfully translated to a distributed FAME-1 simulator; while it lacks any optimizations, the LI-BDNs modeling the blocks targeted for optimization are themselves still labeled with the appropriate annotations. This allows the *LI-BDN transformation* associated with the new feature to either iteratively refine the baseline implementations—as suggested in the original presentation of LI-BDN simulators [89]—or to replace it entirely.

4.6 Summary

The Golden Gate toolchain is a multi-layered software stack that compiles FIRRTL circuits to LI-BDN simulators. It is capable of arbitrarily partitioning designs and compiling each partition to a primitive LI-BDN implementation. For simplicity, early phases rely on target-level transformations to heavily mutate the topology of the circuit; this preserves logical equivalence of the circuit with the input specification, enabling in- or out-of-band checks. Finally, the compiler supports the development of new optimizations. This capability relies on the core infrastructure to extract and decouple optimization targets before optimization-specific transformations iteratively refine the primitive LI-BDN implementation.

Chapter 5

Optimizing Multi-Ported Memories

As outlined in the preceding chapters, Golden Gate provides infrastructure for partitioning a target design and compiling it into a Latency-Insensitive Bounded Dataflow Network (LI-BDN) of communicating simulation components. While the basic Golden Gate compiler flow translates designs into decoupled simulators to allow for flexible co-simulation with software or abstract hardware models, simulation performance does not inherently improve via such modularity. Instead, the main benefit arises from specialization: specifically, the ability to rely on different strategies to translate different partitions of the design to distinct LI-BDN components, or *simulation models*.

Section 4.4 outlines an algorithm for translating a partition of a Synchronous State Machine (SSM) to a primitive LI-BDN. However, it is worth noting that this algorithm could be applied to the entire portion of the target system that is described as a register-transfer level (RTL) design. Indeed, absorbing more of the system into a larger primitive LI-BDN would often result in a net reduction in overhead, as partitioning requires additional queues and finite state machine logic to manage the larger number of I/O token channels. If splitting the simulation of the RTL of a target system is to be useful in practice, it must provide some concrete advantage that outweighs this overhead.

In this thesis, we show that the LI-BDN abstraction enables flexible, composable optimizations that allow larger systems to be simulated on existing FPGA platforms. While these optimizations rely on the more general concept of *decoupling* to trade simulation performance for improved capacity, they leverage the "compositional simulator" design in multiple ways. By dividing the design into components and heterogeneously applying optimizations, Golden Gate takes advantage of specialization, as each optimization is applied where only where it will produce a favorable tradeoff. Furthermore, the LI-BDN structure provides a convenient way to add a further layer of decoupling in the presence of optimizing transformations that save FPGA resources at the expense of simulation speed, as higher-performance components of the simulator can be decoupled from the lower-performance, resource-optimized components. Finally, from a practical perspective, the division of the simulator into separate models simplifies the implementation of optimizations, as some optimizations may act as "silos" that act upon a limited domain of hardware components,

relying on this limited scope to construct highly specialized LI-BDN models. In this chapter, we examine one such optimization that radically reduces the FGPA resource footprint of a particular class of random-access memories (RAMs) commonly found in realistic target systems. By enabling this RAM optimization, significant gains in total simulation capacity can be realized.

5.1 Multi-Ported RAMs

Modern SoCs generally include complex on-chip memory hierarchies that span the range from a processor core's register file to embedded DRAM, with each level defined by a carefully chosen set of parameters. In this section, we focus on classes of memories that are ubiquitous in both the inner memory hierarchy and the microarchitectural implementation of these systems: static random-access memories (SRAMs), register files, and register arrays. At the RTL level, these memories present a simple interface where a set of ports can be issued commands which complete after a fixed latency. Along with width (the size of an entry), depth (the number of entries), and latency, the configuration of these ports is a critical architectural parameter; typically, a memory is defined as having m read ports and n write ports, though some port configurations may include read-write ports that may dynamically perform either read or write operations.

In modern ASIC design processes, these memories may be implemented with a variety of underlying digital circuit elements. Large memories of multiple kilobits, such as the physical memories arrays underlying caches, are typically implemented as SRAMs, which employ a *synchronous read* model where read operations return their result after one or more cycles of read latency. As the area per bit is multiplied by a large capacity, the need for resource-efficient bitcells generally implies a limited port count, with practical designs focusing on single- or dual-ported memories [18]. When architects require larger port counts for memories of significant size, they generally turn to specialized register files. Though most register file implementations could be classified as SRAMs, they employ specialized bitcells and periphery logic that can provide large numbers of ports with potentially sub-cycle latencies [97]. Furthermore, there are many scenarios where an array of values can be conveniently represented with a memory as a tool of microarchitectural abstraction, but small capacity or large port count renders the resulting memory inefficient or impractical to implement with any available SRAM or register file. In these cases, the memory in the RTL design may simply be translated to a multi-dimensional array of registers during logic synthesis, with read and write ports being implemented via multiplexers and de-multiplexers, respectively.

While these underlying implementations are important tools in the ASIC design space, the eventual mapping of a particular target memory to a digital circuit does not directly manifest in an FPGA simulation of the design. This is a direct consequence of the role filled by FPGA simulation methodologies as a model of the *RTL behavior* of the target design. In a typical ASIC process, the RTL design will contain *RTL models* for various memories

that are logically equivalent at that level of abstraction; in general, these models resemble large register arrays with appropriate logic added to model the port configuration and access latency of the memory. Only later on in the design flow will the RTL models be substituted for concrete implementations. Therefore, in this work, we consider on-chip memories as they are presented to tools such as Golden Gate: as abstract multi-dimensional state arrays detached from their eventual implementation target.

5.1.1 Challenges in Mapping Complex RAMs to FPGAs

Modern FPGAs provide abundant memory resources via large numbers of block RAMs (BRAMs), which typically offer some configuration of a fixed number of ports and may be linked together to implement arbitrarily large memories. However, this flexibility is limited by the underlying capabilities of the BRAM instances, which typically limit both port count and configuration. As a result, multi-ported RAMs in ASIC-oriented designs are a classic culprit of poor resource utilization in FPGA prototypes, as they cannot be trivially implemented via BRAMs and are instead decomposed into LUTs and registers [93]. While using double-pumping, BRAM duplication, or FPGA-optimized microarchitectures [58, 57] can help, Golden Gate can automatically substitute a decoupled model to further reduce resource utilization. This enables a target memory with M asynchronous read ports and N write ports to be implemented by time-multiplexing FPGA-friendly BRAMs. In contrast to the static time-multiplexing described in [72] and [30], our optimization can decouple arbitrary target memories and replace them with an optimized model (described in more detail in Section 5.2) without any constraints on the structure or timing of the target design.

5.2 Model Microarchitecture

The optimized memory model is structured around a dual-port, synchronous read memory that stores the contents of the simulated memory. In a typical host FPGA platform, this memory is ideally suited to implementation as a block RAM, unlike the FPGA-hostile memory that it models. Access to this memory is mediated by an arbiter that selects a maximum of two target read/write requests per host clock cycle; this arbitration is dynamic, based upon when the tokens associated with individual ports arrive on their associated decoupled interfaces. As shown in Figure 5.1, an FSM is associated with each target port; together, this vector of FSMs tracks the model's progress in consuming input tokens, performing BRAM accesses, and producing output tokens.

5.3 Adding the Optimization To Golden Gate

To enable our optimization in Golden Gate, we added an analysis pass that finds annotated RAMs and a generator-based, model-implementation pass that inspects the parameters

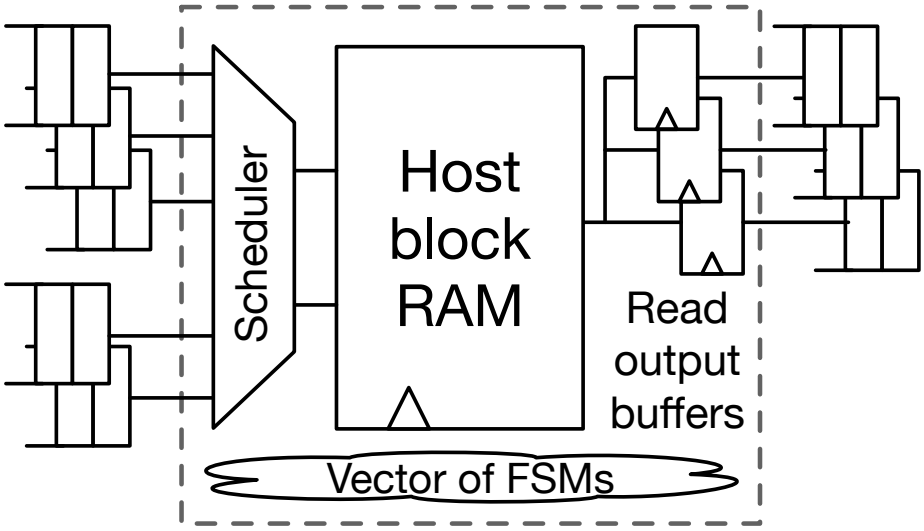

Figure 5.1: A microarchitectural sketch of a three-read, two-write, optimized Golden Gate memory model.

of the target RAM (number of ports, port width, depth) and invokes our model generator (Section 5.2). We also annotated the register file RAMs in our target's source RTL. With these passes enabled, Golden Gate detects and promotes a pair of candidate RAMs for each core of the SoC during target transformation. In simulator synthesis, the implementation pass consumes the RAM modules and produces equivalent models. At this point, the rest of the flow proceeds as described in Section 4.3). Enabling memory substitution adds 5 and 69 seconds of FIRRTL compile time to the quad-core Rocket and hexa-core BOOM configurations, respectively the smallest and largest designs we studied. This is negligible relative to their FPGA compile times of 6 and 22 hours.

5.4 Evaluation

In order to evaluate the multi-ported memory optimization, we look at the relative utilization of different types of FPGA resources—lookup tables (LUTs), block RAMs, and DSP blocks—obtained by enabling the optimization in Golden Gate for several Rocket Chip configurations. Furthermore, since the serialization of memory accesses across multiple FPGA clock cycles increases the latency to simulate a single cycle of each resource-optimized combinational memory, we observe the degradation in performance in the optimized simulator relative to the baseline simulator.

However, while these continuous metrics—utilization and performance—give some insight into the tradeoff of resource efficiency and performance, they do not tell the complete

story. While it is possible to obtain post-synthesis utilization values for arbitrary designs in FPGA tools such as Vivado, users generally target known FPGA devices with fixed resource capacities. If a design fails to successfully complete place-and-route or cannot close timing, the achievable simulation performance is effectively zero. Therefore, the evaluation highlights the ability of the optimization to yield Golden Gate simulator implementations of Rocket Chip designs that would otherwise be impossibly large to simulate on the test host platform, an AWS F1 instance.

5.4.1 Applying the Optimization to Rocket Chip

To motivate the need for this optimization, we use Golden Gate to replace the register files in the application processor cores of several multi-core SoC instances produced by the Rocket Chip Generator [3] described in 2.4. Here, we study two different "core complexes" to explore the relative impact of optimizing register files across different microarchitectures: a standard Rocket in-order core and a large configuration of the BOOM out-of-order core. In each case, we evaluate the impact of substituting each core's floating point (FP) and integer register files for optimized memory models by annotating them with `MemoryModelAnnotations` that hint to the Golden Gate compiler that each should be simulated by a dedicated multi-cycle model. A representative quad-core Rocket Chip system, in this case a LargeBOOM-based design, is shown with the full set of optimization annotations in Figure 6.10.

While Rocket has relatively simple register files, the BOOM core's unified physical register files are parameterized to reflect desired instruction window sizes and number of parallel execution units. Therefore, we describe the relevant configuration details in Table 5.1 and the associated register-file parameters in Table 5.2.

Rocket	BOOM
"Default Rocket"	"LargeBOOM"
Decoupled frontend	8-wide fetch, 3-wide decode
Completion scoreboard	96-entry reorder buffer
31 Int & 32 FP registers	100 Int & 96 FP physical registers
Single issue	1 FP, 3 Int, 1 Mem issued per cycle

Table 5.1: Key configuration parameters for the two target cores. Both designs utilize the RV64GC instruction set and are integrated with Linux-capable Rocket Chip configurations.

5.4.2 Experimental Results

To evaluate the potential savings in FPGA resource utilization from targeting register files with the multi-ported memory optimization, we examine a range of FireSim configurations for multi-core target systems, both with and without the optimization enabled. In each

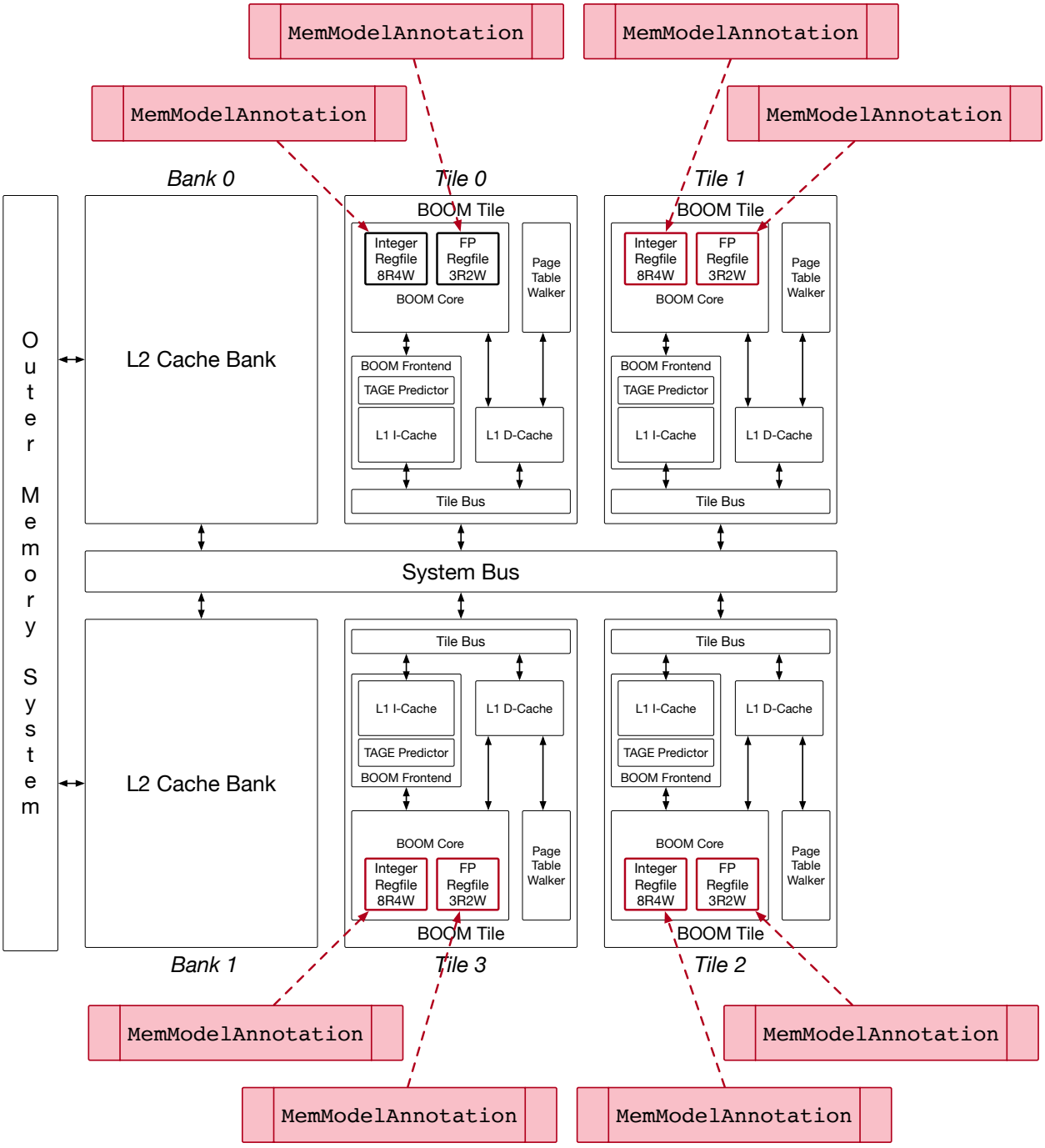

Figure 5.2: A hierarchical block diagram of a quad-core, LargeBOOM-based Rocket Chip system. Here, each register file is labeled with a `MemoryModelAnnotations`, providing a hint to the compiler that each is a beneficial target for the multi-ported memory optimization.

Type	Size	Read Ports	Write Ports
Rocket integer	31 × 64b	2 comb.	1
Rocket FP	32 × 64b	3 comb.	2
BOOM integer	100 × 64b	8 comb.	4
BOOM FP	96 × 64b	3 comb.	2

Table 5.2: Register file specifications for the two target cores. Each is replaced with an optimized model; resource utilization and simulation speed is compared with the baseline FPGA mapping.

case, the target design is a Rocket Chip system with a varying number of LargeBOOM cores, and both the integer and floating-point register files of each core are annotated with a `MemModelAnnotation`. After compilation with Golden Gate, each simulator was synthesized and optimized using Vivado 2018.3.

Figure 5.3 compares the overall number of both LUTs and BRAMs required to synthesize the optimized and unoptimized simulators. While these post-synthesis results omit both place-and-route and timing closure, and therefore do not directly represent the feasibility of implementing a given simulator, they offer useful insight into the achievable level of resource savings. In all cases, replacing the cores' register files with multi-cycle models yields a very large savings in overall LUT count. As the number of cores increases, the fraction of the simulator dedicated to simulating the cores increases, and these savings become relatively larger, and when simulating a four-core target, the optimized simulator requires 25% fewer LUTs than the baseline. Furthermore, the optimization actually allows an even larger increase in simulation capacity than would be expected from LUT count alone, as FPGA-hostile memories place enormous routing pressure on the entire design. Therefore, while the baseline Golden Gate flow can fit only a two-core system on a Xilinx VU9P device, the register file optimization effectively doubles this capacity, successfully implementing three- and four-core simulators at an undiminished host clock frequency of 50MHz.

As discussed in Section 2.1.3, replacing components of the target with decoupled models will impact the FMR—and therefore the simulation throughput—of the simulator. FireSim, being a decoupled simulator, generally has FMR greater than unity. In particular, FireSim uses a last-level-cache and DRAM model [8] (utilization included in *FireSim Misc.* of Figure 5.3) to implement deterministic simulation of the target's outer memory system using host FPGA DRAM. Table 5.3 lists the FMR measured for each simulator configuration over the course of booting Linux and running a Python sorting benchmark on the target system.

Overall, multi-cycle RAM models increases FMR from 1.62 to 12.1, representing a 7.5x reduction in throughput. Here, observed FMR is a function of the highest port-count model in each target, but does not vary across core count, since the models are not combinationally coupled and therefore execute concurrently. Finally, we note that the FMR penalty of using multi-cycle models may often be less than the performance penalty of partitioning—while using only one FPGA and therefore saving both engineering and equipment costs.

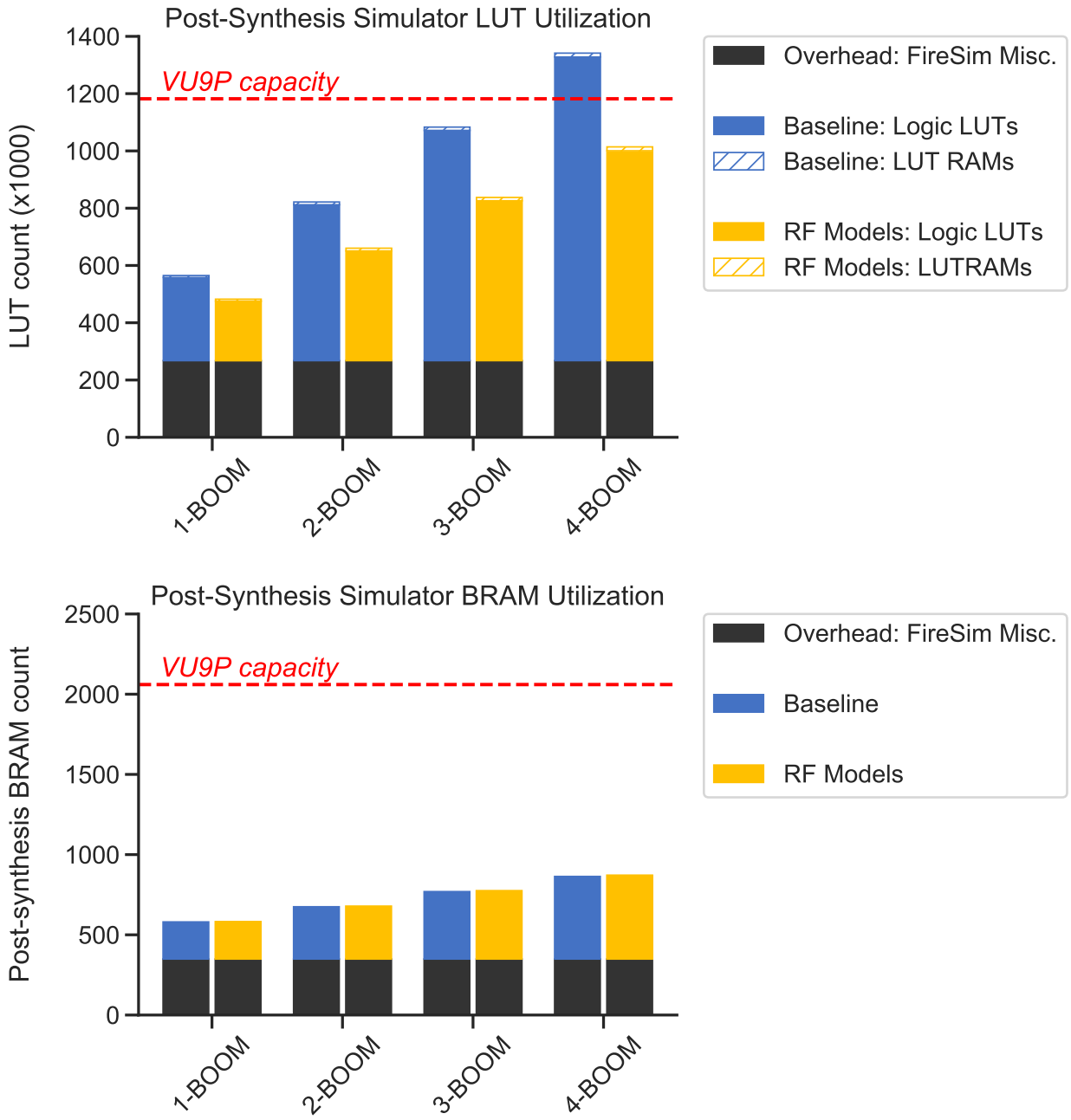

Figure 5.3: A comparison of FPGA resource utilization across baseline and regfile-optimized simulators of multi-core LargeBOOM-based Rocket Chip targets. *FireSim Misc* accounts for all resources in the Amazon-provided shell (v1.4.0) and FireSim hardware for co-simulation; this is fixed across all designs. We omit DSP48s and URAMs as they are constant across all designs and lightly used. *RF Models* three- and four-core BOOM designs failed in placement due to over-utilization–we report post-synthesis utilization. *Optimized* versions of the same designs use 24% fewer logic LUTs and successfully place and route.

FMR	N LargeBOOM Cores			
	1	2	3	4
Unoptimized	1.62	1.62	⊗	⊗
Regfile models	12.1	12.1	12.1	12.1

Table 5.3: A performance comparison of baseline and optimized-regfile simulators. For each simulator configuration that could successfully be implemented on a Xilinx VU9P, observed FMR were collected by booting into Linux and running a Python-based sorting benchmark. Both the 3- and 4-core LargeBOOM configurations failed to successfully route when the register-file optimization was disabled. All feasible simulators closed timing at a 50MHz host FPGA clock frequency, while the ⊗ symbol indicates implementation failure.

For designs that do fit on the FPGA, reduction in routing congestion and register file delay when using the optimized memory model means that host FPGA timing, the other significant driver of simulation throughput, will be no worse than in the optimized design. Here, feasible simulators closed timing at a 50MHz host FPGA clock frequency using Vivado 2018.3 and the default routing strategy prioritizing timing closure. Failing designs were re-attempted with a congestion-optimizing routing strategy, but this did not result in success for any previously infeasible simulator configuration.

5.4.3 Improving Performance with Host RAM Banking

While an 8x slowdown might be an acceptable compromise to avoid falling back on an even slower alternative like partitioning or software RTL simulation, it can result in prohibitively long runtimes for large target workloads. Ultimately, the low bandwidth of the multi-cycle memory model proves to be a severe bottleneck on the rest of the simulator. In order to widen this bottleneck, the host implementation of the model was extended to incorporate multiple banks of underlying FPGA BRAMs.

Figure 5.4 depicts a highly abstracted microarchitecture for a dual-banked implementation of a multi-ported memory model. Here, both banks contain identical data, but each serves half of the target read ports' operations, with write operations broadcast across both banks. In general, this approach resembles techniques used to implement multi-ported synchronous-read memories on FPGAs [58, 57]. However, since these techniques are integrated into a decoupled model, they are used in a novel setting to simulate *combinational*, rather than synchronous reads. Furthermore, the size of traditional multi-ported implementations grows rapidly with increasing number of write ports. Therefore, the banked implementation performs a maximum of two writes per host FPGA cycle, with each bank utilizing an underlying True Dual-Port (TDP) BRAM. This enables a dual-banked implementation to support up to four reads or two writes per cycle. While the enhanced dual-bank scheduler also skips disabled target write operations, read and write operations for a given cycle are not allowed simultaneous access to the host BRAMs, preserving deterministic read-write

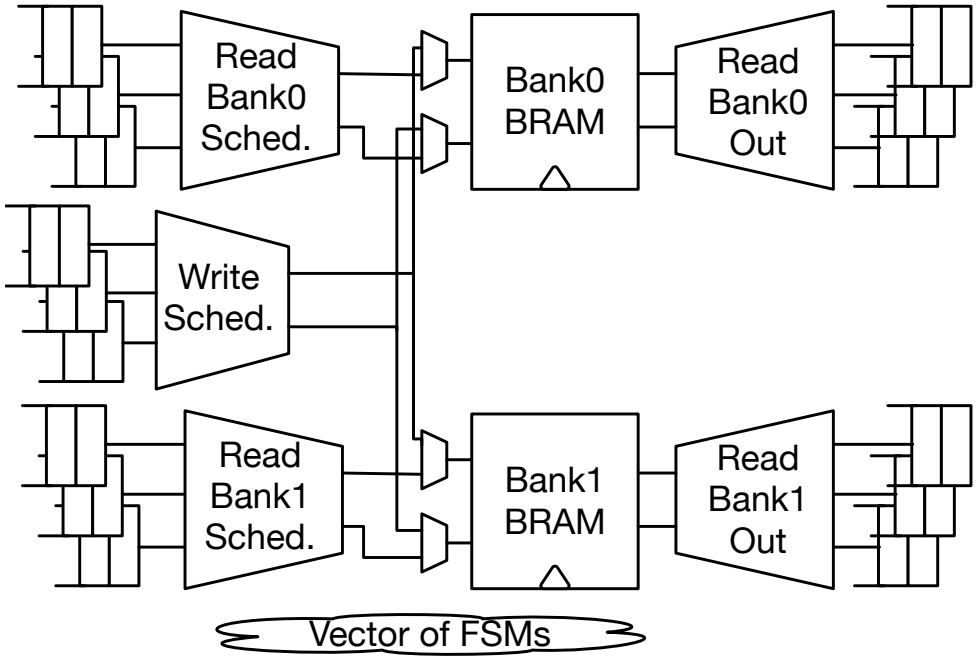

Figure 5.4: A microarchitectural sketch of a dual-banked implementation of a six-read, two-write, optimized Golden Gate memory model.

collision behavior without extra hardware.

FMR	N LargeBOOM Cores			
	1	2	3	4
Unoptimized	1.62	1.62	⊗	⊗
Naive models	12.1	12.1	12.1	12.1
Banked models	5.37	5.40	5.40	5.39

Table 5.4: A comparison of the relative performance impact of multi-ported memory optimizations with naive and dual-banked model implementations. The ⊗ symbol indicates implementation failure.

By employing a dual-banked implementation of the model, the Golden Gate optimization may lower the performance impact of the register-file optimization at the cost of increased utilization. Table 5.4 compares overall simulator performance for configurations that differ only in which variant of the multi-ported memory model was employed by Golden Gate. The banked implementation significantly lowers the observed FMR from 12.1 to 5.4, corresponding to a 125% increase in simulation throughput and a far smaller 3.3x slowdown over the baseline. Furthermore, this implementation resulted in no observable increase in LUT count per model, with run-to-run variation accounting for far larger

changes in overall simulator LUT utilization. However, the use of banked memories does require a larger number of host BRAM macro instances: for the LargeBOOM register files, each banked model used 4 underlying BRAMs rather than the single BRAM of the naive model. This four-fold increase is higher than might be expected given the duplication of data across only two banks, as the naive model allows Vivado to employ a configuration of the BRAM macro that reduces vertical waste. However, since Figure 5.3 illustrates that BRAM utilization is generally a very slack constraint in simulator implementation, banking effectively provides a "free" improvement to the performance-utilization tradeoff.

5.4.4 Summary

Overall, it is clear that this optimization has an outsize impact on simulator utilization. Despite targeting a very narrow subset of the target microarchitecture, replacing FPGA-hostile register files with multi-cycle models can double the number of simulated LargeBOOM cores on a VU9P host FPGA system from two to four. While this does come at a performance tradeoff, the use of banking can limit this slowdown, requiring only 3.3x as many cycles as the baseline simulator to model twice as many cores. Finally, since this optimization requires only a simple hint annotation to be added to arbitrary target designs, it is a broadly applicable tool that can easily be deployed to help designers work around FPGA capacity limitations.

Chapter 6

Optimizing Repeated Instances via Threading

In software systems, where multiple tasks must often contend for a limited set of computational resources, *multi-threading* is a common technique to allocated access in an efficient manner. Rather than serially completing the tasks in sequence, a system may interleave their independent instruction streams—or *threads* of execution—at some finer granularity. Though this does not affect the total work required for the set of tasks, it increases aggregate throughput following a simple observation: the instructions of one task generally do not depend on those of another task; therefore, when a task performs an operation having significant latency and its subsequent, dependent operations cannot be executed, scheduling an alternate thread will allow the system to perform useful work. While this abstract example implies a serial uniprocessor, multi-threading has proven to be a transformative concept in computer systems with widely varying levels of underlying parallelism.

In this chapter, we discuss the applicability of multi-threading to FPGA simulators as a framework for increasing the simulation capacity—i.e., simulating larger chips—by trading off increased execution time for reduced FPGA resource utilization. Furthermore, by relying on the implementation flexibility of decoupled, Latency-Insensitive Bounded Dataflow Networks (LI-BDNs) and the compiler infrastructure provided by Golden Gate, we introduce a novel system to automatically multi-thread repeated instances of large hardware blocks in target RTL designs. By incorporating this transformation as an optional optimization, the Golden Gate simulator compiler may rely on a single underlying threaded model to simulate the full set of instances. While such a model consumes more resources than a direct mapping of a single instance, it is far smaller than the full set, while the use of multi-threading increases per-resource efficiency by reducing the number of idle states in a decoupled simulator. Finally, we demonstrate the utility of this technique in enabling large, multi-core SoCs to be simulated on a single commodity FPGA.

6.1 Multi-Threaded FPGA Simulation

Given the broad applicability of multi-threading, it is not surprising that it has appeared as a feature of multiple FPGA-based simulation systems. In contrast to one classical view of multi-threading, where it is used to maximize utilization of a particular architecture, employing the technique in an FPGA simulator goes hand in hand with an interesting architectural shift: reducing underlying implementation parallelism.

Though a direct FPGA prototype effectively parallelizes all work required to simulate a time step of a target system, this comes at the cost of high utilization of scarce FPGA resources. In theory, this work could be divided into multiple serial sub-steps, each requiring a smaller number of physical resources, effectively reducing parallelism in order to decrease resource utilization. However, since the expensive reprogramming model of FPGAs provides less dynamic flexibility than a traditional programmable processor, common structure must be exploited across the sub-steps to avoid wasting idle hardware. Therefore, previous work in resource-efficient FPGA simulation generally applies such *serialization* to repeated blocks in the target system. The ProtoFlex [22] simulator used a threaded CPU model to simulate in excess of 1000 SPARC processor cores; however, it relied on handwritten models and relegated the emulation of some complex behaviors to software. The RAMP Gold [83] combined a similar approach with an even greater focus on host implementation performance; by employing a separate threaded, functional SPARC execution model and an efficient timing model, a 64-core system could be simulated without delegation of any instructions to software models.

Drawing inspiration from this previous work, we present a multi-threading optimization for Golden Gate. In contrast with previous threaded simulators, which are built around intrinsically threaded models hand-written specifically for a particular simulator, this optimization is automatically applied to arbitrary input RTL designs as a compiler transformation.

6.2 Enabling Multi-Threading in Golden Gate

In keeping with the modular nature of Golden Gate, the multi-threading optimization relies on multiple existing transforms and adds several more. Figure 6.1 depicts the overall process of generating a threaded model from a target design with repeated instances. In this example, two Rocket Chip tiles containing BOOM cores are extracted and mapped to a single threaded implementation.

While threading of models is a property of the host FPGA simulator, Golden Gate attempts to simplify optimizations by modifying the topology of the target design to resemble the topology of the desired simulator. In this case, since the two tiles will be simulated by a separate, threaded model, they are first extracted from the target design hierarchy to separate "islands" at the top level; the semantics of the Golden Gate API imply that each top-level block will be transformed into a separate simulation model. This process relies on the existing

`ExtractModels` transform to modify the target circuit while maintaining logical equivalence; when threading is enabled, all instances annotated with a `ModelThreadingAnnotation` will be extracted to the top level.

After the target transformation is complete, the circuit is transformed to a baseline decoupled simulator. As discussed in Chapter 4, each top-level block will be transformed into a primitive LI-BDN model, and the various models will be composed into a larger LI-BDN simulator. As shown in Figure 6.1, this results in the two top-level instances of the tile module being transformed into two instances of the same primitive LI-BDN.

At this point, the set of models corresponding with the original set of duplicate instances must be threaded. Since they are still instances of a common module, and since the `ModelThreadingAnnotation` has propagated to label the model instances, it is straightforward to analyze the top-level module and find sets of model instances to thread. Using this analysis, the simulator is transformed in two steps. First, a threaded model capable of simulating the number of instances in the set is generated by transforming the FIRRTL implementation of the unoptimized model. Finally, the set of model instances is replaced with a single instance of a threaded model, and their original channel connections are appropriately routed to and from the new model.

6.3 Generating a Threaded Model

Though the focus of this chapter is on threading decoupled simulation models, the transform used to generate the FIRRTL implementation of the threaded models is far more general. Rather than relying on the LI-BDN properties of the input model, it may be described in terms of an arbitrary synchronous transition system. Here, we consider both the abstract structure of the transform along with the low-level strategies used to generate a resource-efficient microarchitecture in the resulting threaded model.

6.3.1 Derivation

Consider a baseline transition system TS, here defined as a Mealy machine on input alphabet Σ and output alphabet O. For simplicity, we will omit the definition of an initial state, and assume that the machine merely begins in any valid state.

$$\begin{aligned}
\text{TS} &= \{\Sigma, \text{O}, \text{Q}, \delta, \lambda\} \\
\Sigma&\text{: the alphabet defining potential input symbols} \\
\text{O}&\text{: the alphabet defining potential output symbols} \\
\text{Q}&\text{: the finite set of states} \\
\delta \in \text{Q} \times \Sigma \rightarrow \text{Q}&\text{: the transition function that defines the next state} \\
\lambda \in \text{Q} \times \Sigma \rightarrow \text{Q}&\text{: the output function on current state and current input}
\end{aligned}$$

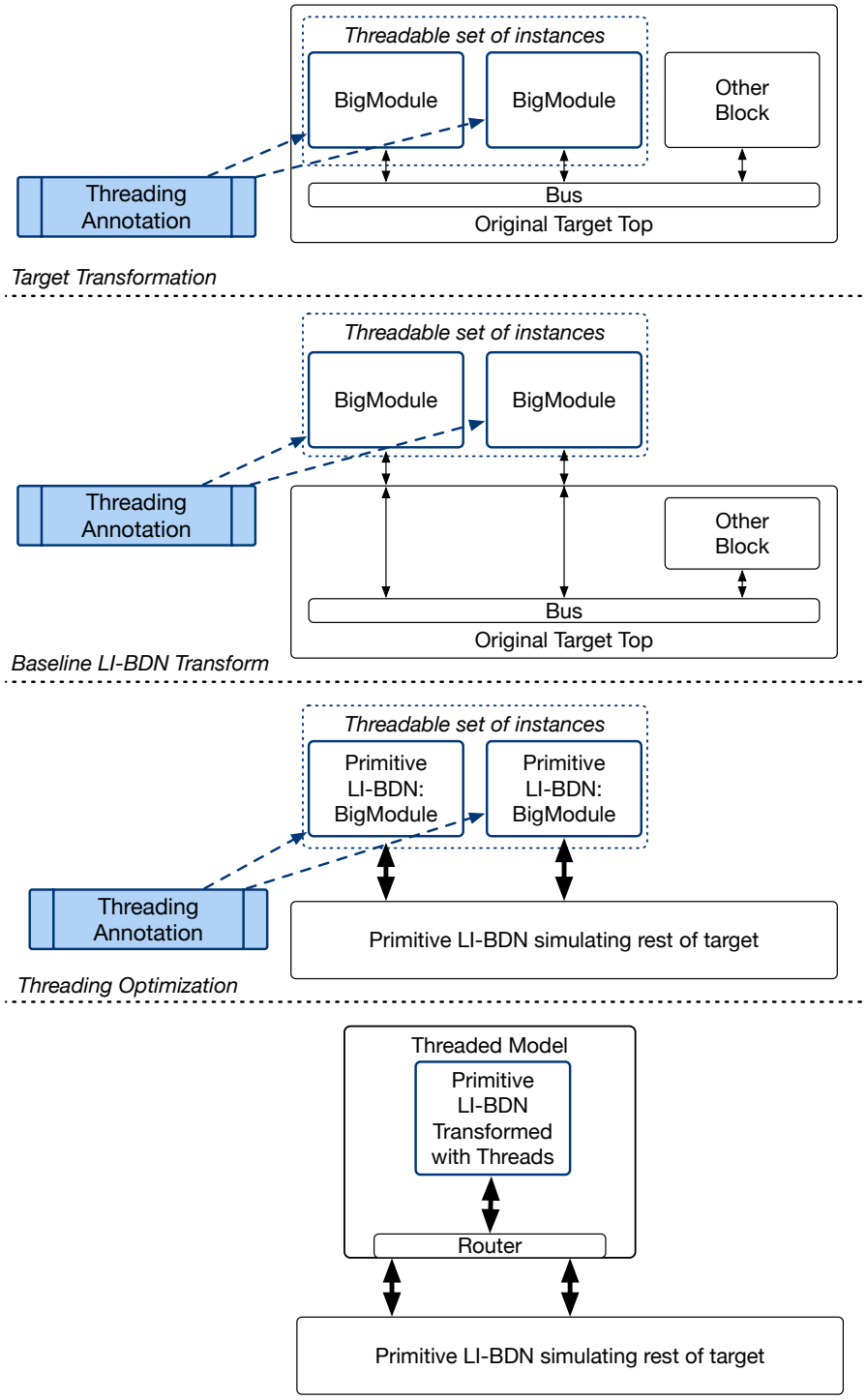

Figure 6.1: A high-level overview of how threading is introduced to a Golden Gate simulator. As shown in Figure 4.5, the optimization depends on two components: the addition of annotations to denote which instances to thread, along with a transform that iteratively refines a set of labeled primitive LI-BDNs to a single threaded implementation.

Now consider a set of identically specified transition systems $\{\mathrm{TS}_1, \cdots, \mathrm{TS}_N\}$, each operating on the same input alphabet Σ and output alphabet O. Note that each system has independent current state q_k, input symbol i_k and output symbol o_k, and the state machine is governed by the following composite behavioral specification.

$$\{\mathrm{TS}_1, \cdots, \mathrm{TS}_N\} = \{\Sigma, \mathrm{O}, \mathrm{Q}_1, \cdots, \mathrm{Q}_N, \delta_1, \cdots, \delta_N, \lambda_1, \cdots, \lambda_N\}$$

$$\begin{aligned} \forall k \in [1, N]\ q'_k &= \delta_k(q_k, i_k) \\ \forall k \in [1, N]\ o_k &= \lambda_k(q_k, i_k) \end{aligned}$$

Since the transition systems are identically specified—though independent—their set of possible states, transition function, and output functions are necessarily identical. Therefore, the preceding set may be further simplified. However, the state transitions for all of the individual systems still proceed in parallel, as indicated by the new behavioral specification.

$$\{\mathrm{TS}_1, \cdots, \mathrm{TS}_N\} = \{\Sigma, \mathrm{O}, N, \delta, \lambda, \mathrm{Q}\}$$

$$\begin{aligned} \forall k \in [1, N]\ q'_k &= \delta(q_k, i_k) \\ \forall k \in [1, N]\ o_k &= \lambda(q_k, i_k) \end{aligned}$$

Now, we may define the notion of a threaded system TS^* capable of *simulating* all of the individual transition systems. However, rather than allowing it to have an arbitrary transition function in $\mathrm{Q}^N \times \Sigma \to \mathrm{Q}^N$, which would include parallel implementations, we introduce a new quantity x, representing the index of the active system, and amend the definition of the system with a function τ for updating the index. Now, we may define a final behavioral specification for the threaded system.

$$\begin{gathered} \mathrm{TS}^* = \{\Sigma, \mathrm{O}, N, \mathrm{Q}, \tau, \delta, \lambda\} \\ \tau \in [1, N] \to [1, N] \text{: the thread scheduling function} \end{gathered}$$

$$\begin{aligned} \forall k \in [1, N]\ q'_k &= \begin{cases} q_k, & i \neq x \\ \delta(q_k, i), & i = x \end{cases} \\ o &= \lambda(q_x, i) \\ x' &= \tau(x) \end{aligned}$$

With this definition in place, it is clear that the threaded variant uses only one invocation of δ and λ for a given transition, despite allowing all states in the full space Q^N. This provides the abstract framework for producing a resource-efficient threaded implementation.

6.3.2 Implementation Overview

As described in the previous section, the threaded implementation may rely on the same transition relation δ as the original input system. From a FIRRTL compiler perspective, this implies that the combinational logic of the system—the concrete implementation of this function—may generally remain unchanged. However, the behavioral rule by which the state of the system is updated with the application of this transition function no longer applies. Since this behavior is implicit to the definition of the input system, the design must be modified to ensure that only the state of the active thread is passed to the state transition function or modified in a particular cycle. Furthermore, an implementation of the scheduling function τ must be added to the transformed implementation.

In practice, this modification is most easily performed by modifying the structures in the design that contain state: registers and memories. By expanding their size to accommodate a vector of N states, and by adding the appropriate logic to select and update the appropriate element of this vector, the threaded implementation may be generated through small peephole changes to the input RTL.

A high-level outline of this transformation is included in Figure 6.2. One key distinction between an abstract transition system and a concrete FIRRTL implementation is the existence of a module hierarchy. Fortunately, though an input model may contain submodules, its entire specification may be transformed by transforming all modules that appear in its hierarchy from bottom to top, and by appropriately substituting instances for the resulting threaded modules. This requires some bookkeeping, as the same module may appear in both threaded and un-threaded contexts, or even in two contexts with different degrees of threading. Therefore, the transform must transform a copy of the input module rather than the original.

6.3.3 Input Circuit Preconditions

In keeping with the general FIRRTL philosophy of dividing hardware compilers into self-contained layers, the model-threading transform of Golden Gate is relatively isolated from the rest of the compiler. Rather than relying on deep introspection into the concrete implementation of decoupled models, it is generally agnostic to the behavior of the upstream FAME-1 transform that generates the unoptimized LI-BDNs it later optimizes.

Given this generality, the multi-threading transform can multi-thread a nearly arbitrary input circuit. This operation is defined in terms of a simple host environment, assuming only the existence of an ungated main host clock. Any block that meets this criterion may be *host multi-threaded* N ways; the block may include arbitrary I/O, combinational logic, and state, and may even rely on clock gating internally to enable or disable updates to state elements or memories, as long as all such gated clocks are synchronously derived from the main clock. This not only supports baseline LI-BDN models that rely on clock gating, but even allows optimized, hand-written variants such as the multi-cycle RAM model to be multi-threaded.

```
def thread_model(model_module, N):
  threaded_modules = {} // cache for transformed modules
  def thread_module(module, N):
    assert module has host clock port
    add thread_idx counter
    for each state-containing structure:
      replace with a threaded implementation
    for each RHS reference using a state-containing structure as a driver:
      update to retrieve data from the threaded implementation
    for each connection updating a state-containing structure:
      route the RHS data to the write port of the threaded implementation
    for each submodule instance i:
      if !threaded_modules.contains(i.module.name):
        threaded_modules[i.module.name] = thread_module(i.module, N)
      replace with instance of threaded_modules[i.module.name]
    return transformed module
  return thread_module(model_module, N)
```

Figure 6.2: Pseudocode for threading a FIRRTL implementation of a decoupled model.

While such a transformation would significantly reduce the performance of the simulator, it has proven useful as a "stress test" for the robustness of the Golden Gate compiler.

Although the multi-threading transform is generally applicable to arbitrary models of arbitrary target designs, it does rely on standard FIRRTL lowering infrastructure. Before a model is threaded, it is lowered to Low FIRRTL and augmented with standard type and drive direction analyses. Furthermore, threading depends on successful identification of repeated instances of the same un-threaded model; this process is aided by various de-duplication transforms that merge the definitions of modules that contain functionally identical implementations.

6.3.4 Thread-Management Logic

At its core, multi-threading saves FPGA resources by allowing the simulation of a single cycle of set of instances to be serialized over multiple cycles. This structure comprises a transition system containing the state of all the instances, but a simpler transition function that affects the state of only a single *active* instance.

In the concrete implementation, the multi-threaded model must contain scheduling logic responsible for selecting the thread representing the active instance. In general, this scheduling logic may be classified into one of two categories: *static*, where the order of the threads is fixed, and *dynamic*, where it is a function of the environment or external I/O. In particular, while a static schedule for N instances iterates through them in a round-robin order over N host FPGA cycles, a dynamic schedule might attempt to preferentially select an instance with all input tokens available on a given cycle, as it would be more likely to make forward progress and advance the target simulation of that instance.

While dynamic scheduling offers theoretical advantages in simulation throughput, it comes at significant implementation cost, as arbitration decisions must be broadcast across the threaded model. Furthermore, it is important to note that the "workload" faced by the simulator is generally low in dynamic behavior: whether or not a particular instance—say, a particular core—is idle on a given target cycle, it must still be simulated. Though some components of FireSim, such as the network model, present dynamic and unpredictable host timing in response to target behavior, these components are generally not tightly coupled with individual cores or other prototypical targets for multi-threading. Therefore, we rely on a static schedule for its extremely low implementation overhead.

6.3.5 Threading State Elements

With arbitration of I/O channels and thread scheduling handled at the boundary of an individual threaded model, the core implementation of the threaded model can be generated by transforming all state in the input design to an array of sub-states and a means to ensure that only the active sub-state is updated on a given host FPGA cycle. However, in order to make this process amenable to a source-to-source FIRRTL transformation, we do not directly materialize any monolithic representation of either the state or the transition function. Instead, the transform replaces each instance of a structure containing state in the input circuit—i.e., a register or a memory—with a structure that manages sub-state selection for the set of state bits contained within the input structure.

In keeping with the overall goal of conserving FPGA resources, Golden Gate specializes this process for each of the three common types of state-containing structures in target RTL designs: registers, asynchronous-read memories, and synchronous-read memories. Fortunately, this process is greatly simplified by the inclusion of first-class register and memory primitives in the FIRRTL language. In contrast with a Verilog-based intermediate representation, which would require complex and imperfect algorithms to infer which variables correspond with each type of structure, FIRRTL directly exposes all relevant information, including clocking, port structure and read & write latency. With the support of these language features, each type of structure is transformed via a microarchitectural pattern that efficiently implements the array of sub-states and the appropriate selection mechanisms.

Registers

While they collectively contain relatively few bits compared to large caches and other memories, individual registers are by far the most numerous state elements in the input designs. Though FIRRTL supports registers of aggregate types, such as bundles or vectors, the use of standard FIRRTL lowering passes as prerequisites of multi-threading implies that registers contain only signed and unsigned two's complement types, further increasing the number of declared registers. Therefore, it is imperative that hardware resulting from transforming these registers have not only minimal per-bit cost but also minimal overhead per instance. To this end, Golden Gate includes two alternative flows for threading basic multi-bit reg-

isters, with each flow converting a register to different structure containing the set of "copies" of the register in the different simulated instances.

In the first flow, each register is replaced with a small, asynchronous-read memory of the same data type with a single read and write port. By using the same clock that drove the original register, updates to the contents of the memory will be appropriately disabled under the same clock-gating conditions, while its asynchronous reads will be unaffected by clocking. This memory has depth equal to the number of threads N, and is addressed for both read and write operations by a wrapping, free-running counter. While the instantaneous value of this counter is a function of the current thread index, it is important to note that the address can be generated via an arbitrary bijection with the "true" thread index. This eliminates any requirement either reset or synchronization of this address value across all threaded registers in the generated model, allowing an arbitrary number of replicated address counters to reduce fanout. As shown in Figure 6.3, this transformation relies heavily on the provision of first-class register and memory objects in FIRRTL. Furthermore, it uses the built-in analyses discussed in Section 4.2.3 to help classify references to the original register, substituting the data field of the new memory's read port where the register was used as a driver and the data field of the write port where it was being driven. As the read port instantaneously outputs the current value held in the "slot" corresponding with the current thread, the enable signal of the read port is tied high. However, since the input design could potentially rely on both conditional register connections and clock gating to control when the register's value might change, the transformation must ensure that the contents of the memory at the current thread's address must follow identical state transitions under identical conditions. While the use of the register's original clock ensures appropriate behavior under clock gating, FIRRTL semantics allow multiple solutions to the "conditional update" problem. Fortunately, a simple concrete solution consists of replacing conditional register updates with conditional connections of the right-hand side to the new memory's write data input and conditional assignment of a set bit to its write enable. However, while the register-to-memory flow includes this behavior, as a practical consideration, the FIRRTL lowering phases that precede the multi-threading transform guarantee that all conditional updates will have been removed from the input design.

While the register-to-memory flow is sufficiently general to apply to all FIRRTL designs, the multi-threading transform can be configured to map input registers with a different strategy, depicted in Figure 6.4. This alternative microarchitecture replaces each register with a shift register and a ring-like feedback path. In contrast with the memory-based microarchitecture, these *circular shift registers* obviate the need to either broadcast or replicate thread index counters. However, since the data must advance on every FPGA cycle to present the appropriate state copy for the active thread, the ring must be clocked with the ungated main host clock. This in turn requires a more elaborate solution to accommodate clock gating in the input design: a single-bit counter is used to recover whether the gated clock was active during the preceding rising edge of the host clock. While these counters pose the same fanout-vs-replication challenges as the thread-index counters of the memory-based approach, they require fewer hardware resources. As demonstrated in Figure 6.4 for $N = 4$,

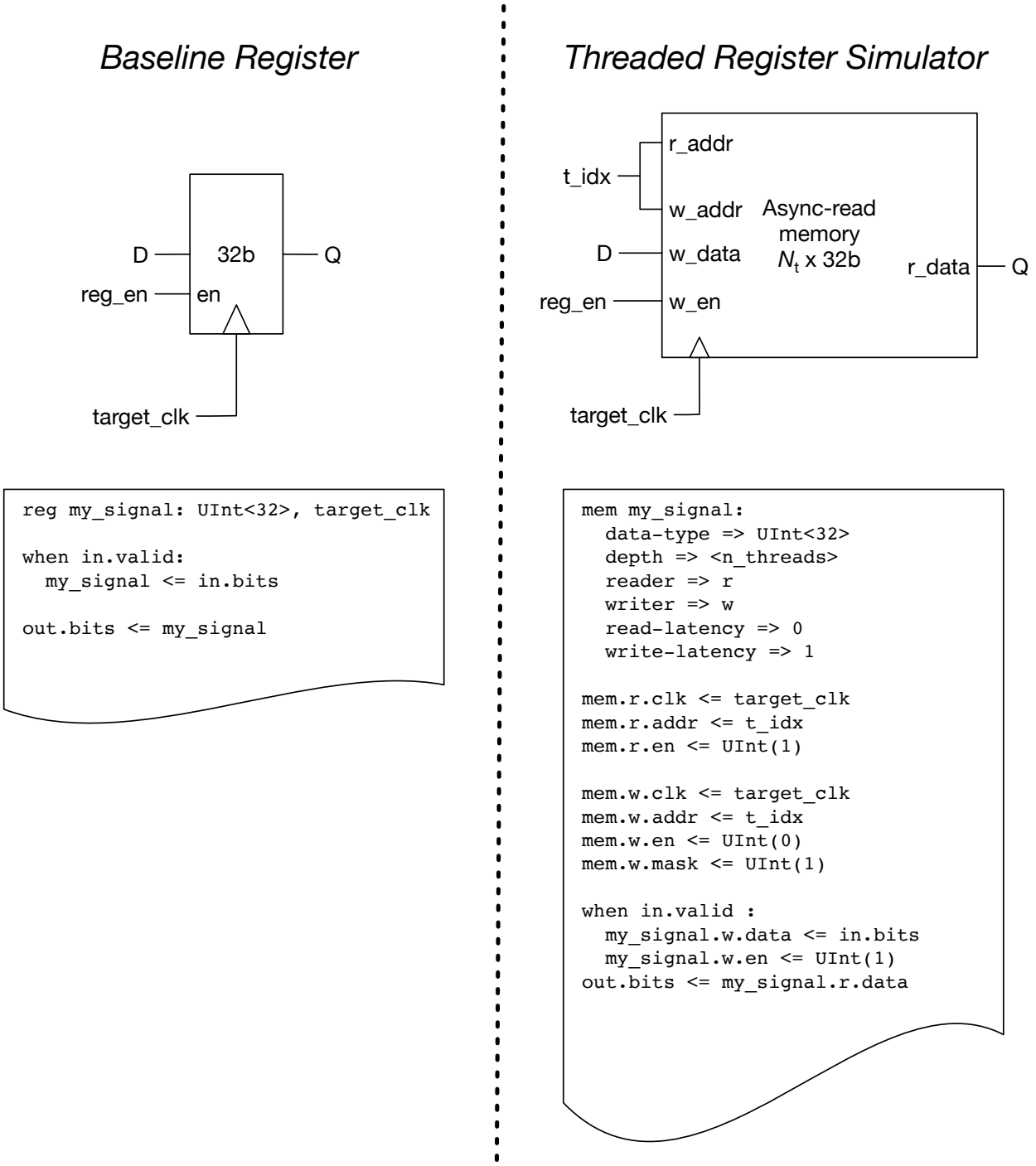

Figure 6.3: A high-level overview of how multiple threads' copies of an individual register may be simulated with a combinational-read memory of depth N_{threads} that is addressed by the index of the active thread. A representative input-output pair of FIRRTL code blocks shows how this structural pattern can be translated to statement-based RTL.

though right-hand-side references that use the original register as a driver are replaced with references to the active slot in the ring, reusing the name of the original register allows this substitution to be performed implicitly with no code changes. Similarly, connections driving the original register are transformed to instead update the register immediately following the active slot in the ring.

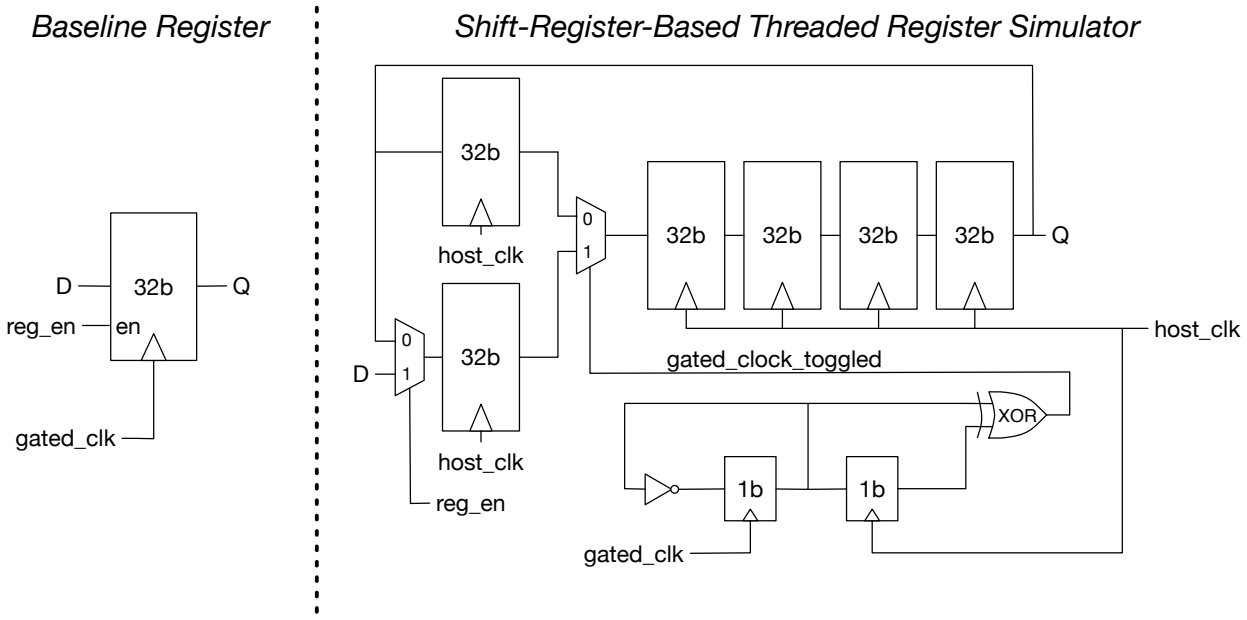

Figure 6.4: A shift-register-based pattern for holding the state of a simulated register in four simulated instances' threads. A specialized component at the bottom of the diagram records recovers the behavior of a gated clock, allowing this transformation to be applied to circuits with multiple synchronous that have independent gating conditions.

With both strategies available, the compiler is free to select one or the other—or, indeed, to mix them at any granularity. While the two techniques are compared in Table 6.1, it is worth considering how each might be implemented in the Xilinx FPGAs targeted by FireSim. Given a reasonable range for degree of threading—say, $N \in [8, 128]$ instances—the microarchitecture in Figure 6.3 will replace an m-bit register with an $N \times m$, single-read, single-write, asynchronous read memory. Such a memory will invariably be mapped to lookup tables in RAM mode (LUTRAMs); while these LUTRAMs consume valuable LUT resources, it is certainly more efficient than an array of flip-flops and multiplexers. While Figure 6.4 initially appears quite different, it too is capable of being implemented efficiently by re-purposing the storage underlying LUTs in the form of Shift Register LUTs (SRLs). However, empirical results with Vivado 2018.3 demonstrated that while the memory-based register-threading transformation will cause efficient inference of LUTRAMs even at low numbers of threads, SRL inference was not achieved for any variation of the shift-register-based transformation. Furthermore, as shown in Table 6.1, the use of small memories led to lower utilization of

both LUTs and flip-flops. Therefore, the memory-based transformation shown in Figure 6.3 is employed by default for the Golden Gate multi-threading optimization.

Register transformation	2-core LargeBOOM		3-core LargeBOOM		4-core LargeBOOM	
	LUTs	*FFs*	*LUTs*	*FFs*	*LUTs*	*FFs*
Small memory	748,844	370,354	801,761	389,371	866,681	413,849
Shift register	780,427	614,898	844,212	717,665	928,436	815,358

Table 6.1: Measured post-synthesis resource utilization counts for lookup tables (LUTs) and flip-flops (FFs) for two different register-threading microarchitectural transformations. For each simulator configuration, an N-core LargeBOOM target system was mapped to a simulator, with the N cores' tiles implemented in a single threaded model.

Asynchronous-Read Memories

As shown in Figure 6.5, asynchronous-read memories are the simplest to thread of all the state-containing structures. The data-type and port configuration of the memory remains unchanged, while its depth is multiplied by the number of threads. The drivers of the ports' address fields are recovered via analysis of the circuit, and the active thread index is prepended to the original value; this effectively defines a translation where address $(n \times \text{depth}) + i$ holds the value corresponding with address i in the original memory for thread context n. When N_{threads} is a power of two, this effective address may be computed by concatenating active thread index with the original address; otherwise, a simple accumulator is necessary.

Though this transform scales the capacity of the memory linearly in the number of threads and therefore consumes the same total number of bits as the memories in the original set of unthreaded instances, it is important to keep the limitations of the host platform in mind. These asynchronous-read memories are generally mapped to LUTRAMs; while they are efficient for moderate sizes up to several kilobits, they scale poorly with extreme depth or complex port configuration. This inefficiency in mapping FPGA-hostile memories—the primary motivation for the memory optimization discussed in Chapter 5—is effectively worsened by threading. However, the compositional nature provides a natural solution: as discussed in Chapter 7, the two optimizations may be combined, allowing those memories not amenable to threading to be simulated via highly efficient multi-cycle models.

Synchronous-Read Memories

In contrast with asynchronous-read memories, whose internal state corresponds exactly with the set of addressable data entries, each read port of a synchronous-read memory implicitly carries extra state: namely, the value read from the memory on the previous rising edge of the clock. Therefore, any microarchitecture for threading these memories must maintain a

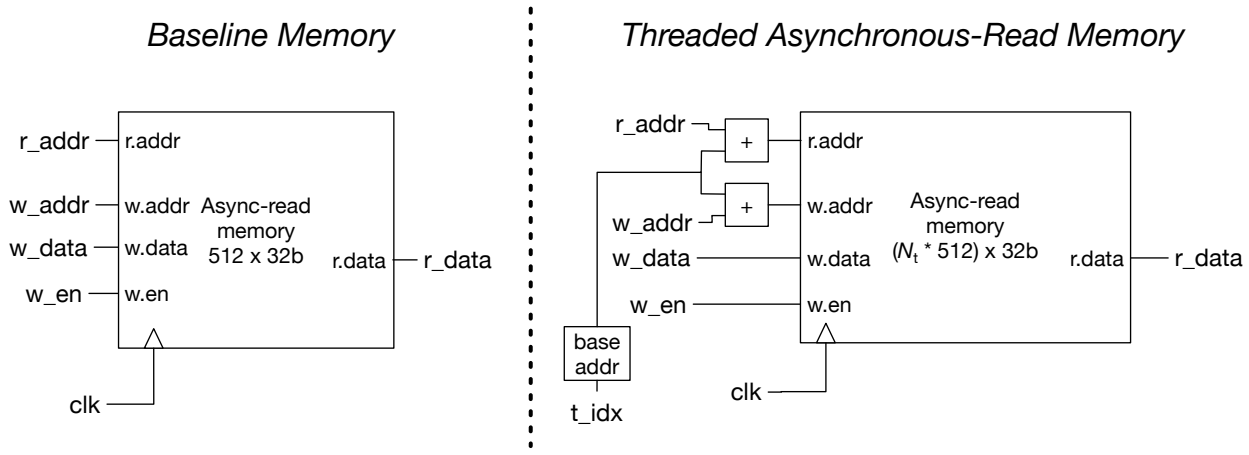

Figure 6.5: An asynchronous-read memory may be threaded by increasing the depth by a factor of N_{threads} and offsetting the active read and write addresses. Here, the original driver of the address is used as an offset from a base address derived from the thread index and memory depth.

copy of the appropriate read port state for each of the threads, which renders the approach of simply increasing memory depth unsuitable without the addition of new structures to maintain this state.

When designing a concrete implementation to meet this abstract specification, it is important to consider the modified timing of a read operation in the host clock domain. In particular, to model a synchronous-read memory with a single-cycle read latency for N threaded instances, the data resulting from a particular read request must actually be returned N cycles later, as depicted for $N = 4$ in the timing diagram in Figure 6.6. As shown in Figure 6.7, the addition of $N - 1$ additional pipeline stages would not only correct achieve this timing, but would provide sufficient additional state capacity for the per-port read data state. Unfortunately, this simple design relies on an ungated host clock to align the output of the memory with the active thread; therefore, it cannot be used to thread memories that are connected to gated clocks in the input design, rendering it unsuitable for transforming arbitrary decoupled models.

In order to maintain the behavior of the original memory when the clock is disabled, a simple implementation could employ N independent memories and further clock gating, disabling the clock for each memory whenever its corresponding thread is inactive. However, this highly inefficient implementation would not only prevent consolidation into a single memory, but would also waste scarce clock-gating resources. Instead, we employ the clock-edge counter circuit introduced in Figure 6.4 to record whether the clock for each read port was active at the previous host cycle boundary. With this information now encoded as a synchronous Boolean control signal, simple microarchitectures can be synthesized to manage the buffering of each thread's read data. As shown in Figure 6.8, this control signal may be used as a write enable for a small memory of depth N, with each entry in the memory

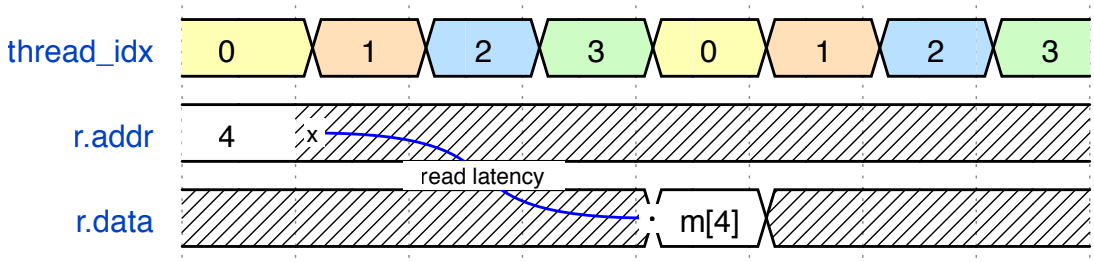

Figure 6.6: A timing diagram depicting how the use of threading effectively extends the allowable latency for read operations on synchronous-read memories. Here, with $N_{\text{threads}} = 4$, the threaded memory implementation has four cycles to return the read data.

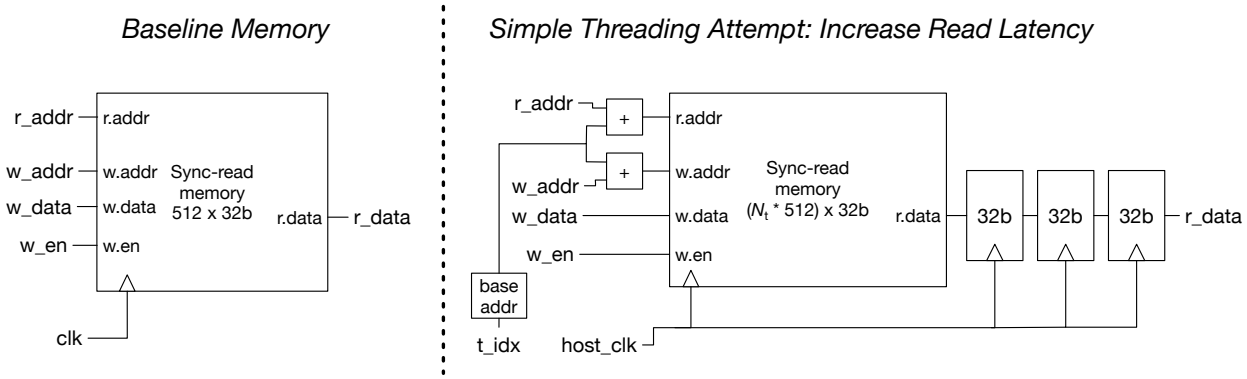

Figure 6.7: While adding extra pipeline stages alone is an appealing solution to thread synchronous-read memories, it does not compose with the use of clock gating on the original memory. This example with $N_{\text{threads}} = 4$ is not applicable to decoupled models that rely on clock gating internally.

containing the current read data state for an individual thread. By generating a memory to act as an N-entry output buffer for each read port, the behavior of each thread may be appropriately simulated even in the presence of gated clocks.

While this output-buffering approach introduces extra hardware, it offers two significant advantages in the resulting FPGA implementation. First, while memories in ASIC RTL may have arbitrary size, synchronous-read block RAMs (BRAMs) provisioned in modern FPGAs have fixed sizes with limited parameterization. While they may be combined to implement larger memories, they are relatively inefficient below a certain minimum depth. Current Xilinx FPGAs have a minimum BRAM depth of 512, which leads to significant "vertical waste" when implementing shallower target memories [94]. By striping the contents of the simulated memories of several threaded target instances across one deeper underlying host memory, many instances of this vertical waste may be avoided. Furthermore, since the results of a read do not need to be made available until N cycles later, it is relatively simple to introduce additional pipelining for sufficiently large values of N. In practice, the pipelined implementation shown in Figure 6.8 is employed for four or more threads, which effectively

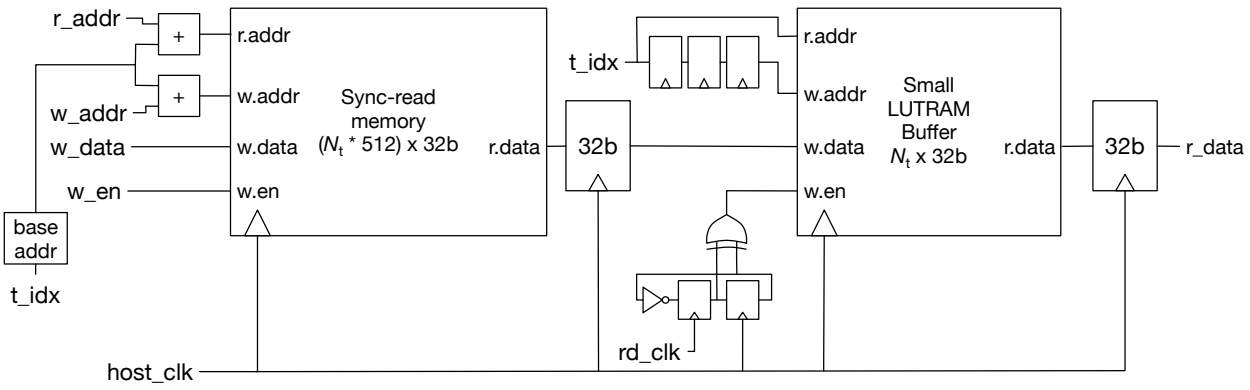

Figure 6.8: A 4-way threaded implementation of a synchronous-read, simple dual port, 512x32b memory. The output buffer stores the data that should appear at the read port data output for each thread, and the associated addressing logic ensures that the result of a read from thread i appears at `r_data` four cycles later when the thread is next scheduled.

removes the output delay of the memory from downstream logic paths.

While these examples depict microarchitectures to simulate a memory with a single read port and a single write port, it is important to note that a buffer must be provisioned for the read data of each read port and each read-write port. As shown in Figure 6.8, the underlying host memory is generated with the same port configuration as the original non-threaded memory: in this case, a single read port and a single read-write port. Since Golden Gate is generally used to model realistic ASIC RTL, target synchronous-read memories are generally limited to a single read-write port or a combination of a dedicated read port with either a write or read-write port; these configurations correspond with common 6T and 8T SRAM architectures, respectively. In practice, this is not a significant bottleneck for modern Xilinx or Intel FPGA platforms, as each provides dedicated block memories that may be configured with up two ports of any type (read, write, or read-write), a broader space than typical target memories [94, 44]. However, it is worth noting that the ability to add multiple additional pipeline stages allows the threaded memory model to have a very short critical path delay, making it possible to double-pump the memory by using both the positive and negative edges of the ungated host clock. Future modifications to the threading transform could incorporate double-pumping and extra buffering structures to model memories with a mix of up to four ports of arbitrary type.

Due to the larger number of components and higher degree of parameterizability in the resulting microarchitecture, the Scala implementation of the microarchitectural substitutions that thread synchronous-read memories are significantly more complex than those for either registers or asynchronous-read memories. However, since a FIRRTL memory has structural ports, we borrow a technique used in other FIRRTL transformations that modify memories: the threading transform replaces synchronous-read memories with black-box instances, and a subsequent transform replaces the black-box modules with concrete implementations. This

not only helps maintain modularity in the software implementation, but since the threaded synchronous-read memory module has a strict superset of the I/O of the original memory, it also limits the changes required to the enclosing module to simply connecting the free-running host clock and current thread index.

6.4 Routing I/O at the Threading Boundary

After generating an N-way threaded implementation to replace a set of N repeated model instances, each channel formerly connected to such an instance must be appropriately routed to the threaded model. Fortunately, the static thread schedule makes this relatively simple. As shown in Figure 6.9, for each input channel of the threaded model, a static round-robin arbiter selects a token from the source of the corresponding channel in one of the original instances. This static schedule is governed by a thread index counter that continuously iterates through the range $[0, N)$. When the thread with index n is selected, the sources of data for the input channels of the n-th instance in the original set are selected and may pass tokens to the input channels of the threaded model. Similarly, while the data from each output channel of the threaded model is broadcast to all n sinks that would have received data from the corresponding channel in the original instances, only the sink originally connected to the n-th instance would be permitted to handshake and receive a valid token in that cycle.

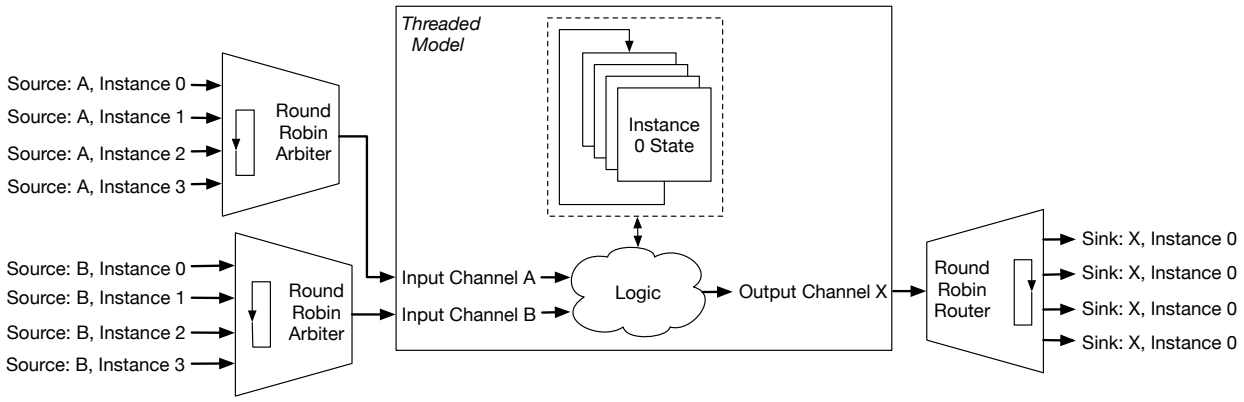

Figure 6.9: A scheme for routing I/O channels to and from a threaded model implementation. Since the threads are statically scheduled, a strict, static round-robin arbiter for each common input channel is sufficient to select the appropriate input sources for the active thread. Similarly, a statically scheduled router for each common output channel can direct output tokens to the appropriate sinks in the enclosing LI-BDN.

Since the routing costs of this scheme scale with the number and bit-width of the I/O channels of the instances selected for threading, it is important to select the appropriate module in the target design to thread. As a heuristic, choosing the highest-level block of the design that is still highly duplicated is a reasonable strategy for reducing the cost of the I/O

routing. Though this may increase the number of channels at the boundary of each repeated instance, the routing constraints that govern realizable ASIC target systems—the primary application domain for FireSim—generally lead to larger blocks having a higher ratio of logic to I/O.

While generating a full, parallel routing infrastructure for selecting tokens by thread is straightforward to implement in a compiler and reasonably resource-efficient, it is not the only way to deal with the I/O boundary between threaded and non-threaded portions of a decoupled simulator. Furthermore, the routers and arbiters that must be instantiated consume a significant number of logic resources. Some previous work has explored alternative schemes for routing I/O to threaded models. In the HAsim project [71], numerous underlying implementations for an on-chip network were synthesized for a simulator employing a threaded, decoupled model of multiple processor cores. Using a technique referred to as *permutation*, the designers effectively managed to thread the queues and routers at the I/O boundary of the threaded model, significantly reducing implementation overhead. However, this technique relies on semantic understanding of the structure of the I/O that enter and exit the threaded model: specifically, it must be interpreted as an on-chip network with a particular structure. While this enables an extremely efficient implementation, this inference is not generally compatible with a general compiler like Golden Gate, which must be able to thread arbitrary RTL with arbitrary unstructured I/O. Therefore, we exclusively rely on the per-channel arbitration scheme when multi-threading models in Golden Gate.

6.5 Evaluation

As with the multi-ported memory optimization discussed in Chapter 5, we evaluate the efficacy of the instance multi-threading optimization by generating FireSim simulators for a set of multiple Rocket Chip configurations. In particular, we explore the impact of increasing core count on simulator FPGA resource utilization and throughput, both with and without the optimization. Finally, we compare the peak capacity of a fixed-size FPGA host platform, measured in number of simulated cores, in order to define the scenarios where applying the utilization-performance tradeoff of the optimization would represent a practical and performant solution for simulating large systems.

6.5.1 Applying the Optimization to Rocket Chip

In a symmetric multi-core SoC, the set of identical processor core instances is a natural target for multi-threading. However, a realistic target will generally have a hierarchy of components associated with each physical core. Figure 6.10 shows the structure these core complexes, or *tiles*, for a Rocket Chip system with four LargeBOOM cores.

Given the desire to maximize the amount of logic reused via threading and to minimize I/O cuts across the system, the collection of tile instances provides the greatest theoretical benefit from multi-threading. However, each tile contains not only the core, but its associated

private L1 caches. As the resource footprint of these caches is dominated by large SRAMs, it is imperative that these memories are implemented efficiently by the threaded model. Fortunately, the threading strategy described in 6.3.5 has minimal overhead and actually removes much of the delay associated with the memory itself from paths involving read and write operations. Even though state elements like SRAMs derive no intrinsic theoretical benefit from threading, the use of a single deeper memory in the threaded model to replace N shallower memories can lead to lower vertical waste when mapped to fixed-size BRAMs, further enhancing resource efficiency. Therefore, since applying the optimization to the full tile with both caches presents the greatest potential benefit, this strategy is used throughout the experimental evaluation.

6.5.2 Experimental Results with Multi-Core BOOM Systems

To evaluate the impact of the instance multi-threading optimization, we compare the FPGA resource utilization and performance of generated FireSim simulators both with and without the optimization enabled. In each case, the target design is a Rocket Chip system with a varying number of LargeBOOM cores, and each `BoomTile` is annotated with an `EnableModelMultiThreadingAnnotation` to label it as a potential target for threading. By adding `MTModels` to the FireSim platform configuration, we may selectively enable threading to produce both optimized and baseline simulators by passing each target system through the standard FireSim and Golden Gate flow depicted in Figure 4.1.

To illustrate the impact of threading on FPGA resource utilization, the results of synthesizing each simulator configuration were obtained from intermediate reports generated by Vivado 2018.3. Figure 6.11 compares the overall number of both LUTs and BRAMs required to synthesize the optimized and unoptimized simulators. While these post-synthesis results omit both place-and-route and timing closure, and therefore do not directly represent the feasibility of implementing a given simulator, they offer useful insight into the achievable level of resource savings. In all cases, instance multi-threading significantly reduces the number of LUTs required by the simulator. Even for the case of a two-core LargeBOOM simulator, where the overhead introduced by threading is poorly amortized over a small number of instance threads, the optimized implementation requires 8% fewer LUTs. Furthermore, the reduction in LUT count grows dramatically with increasing core count: since the threaded model merely extends the number of entries in its state elements to simulate more cores, this advantage grows to 28% at three cores and 35% at four cores.

To better understand these savings, we may further divide this LUT utilization into three sub-categories: resources used by the auxiliary components of FireSim that act as a "shim" to the host AWS instance, and within the actual LI-BDN simulator generated by Golden Gate, either LUTs used to implement logic or LUTs used in RAM mode to implement combinational-read memories. While the miscellaneous FireSim components use a significant number of resources, they are constant across all simulator configurations. Therefore, the reductions in overall LUT count reflect an even larger change in the relative utilization of the LI-BDN implementation. Furthermore, while threading significantly reduces the

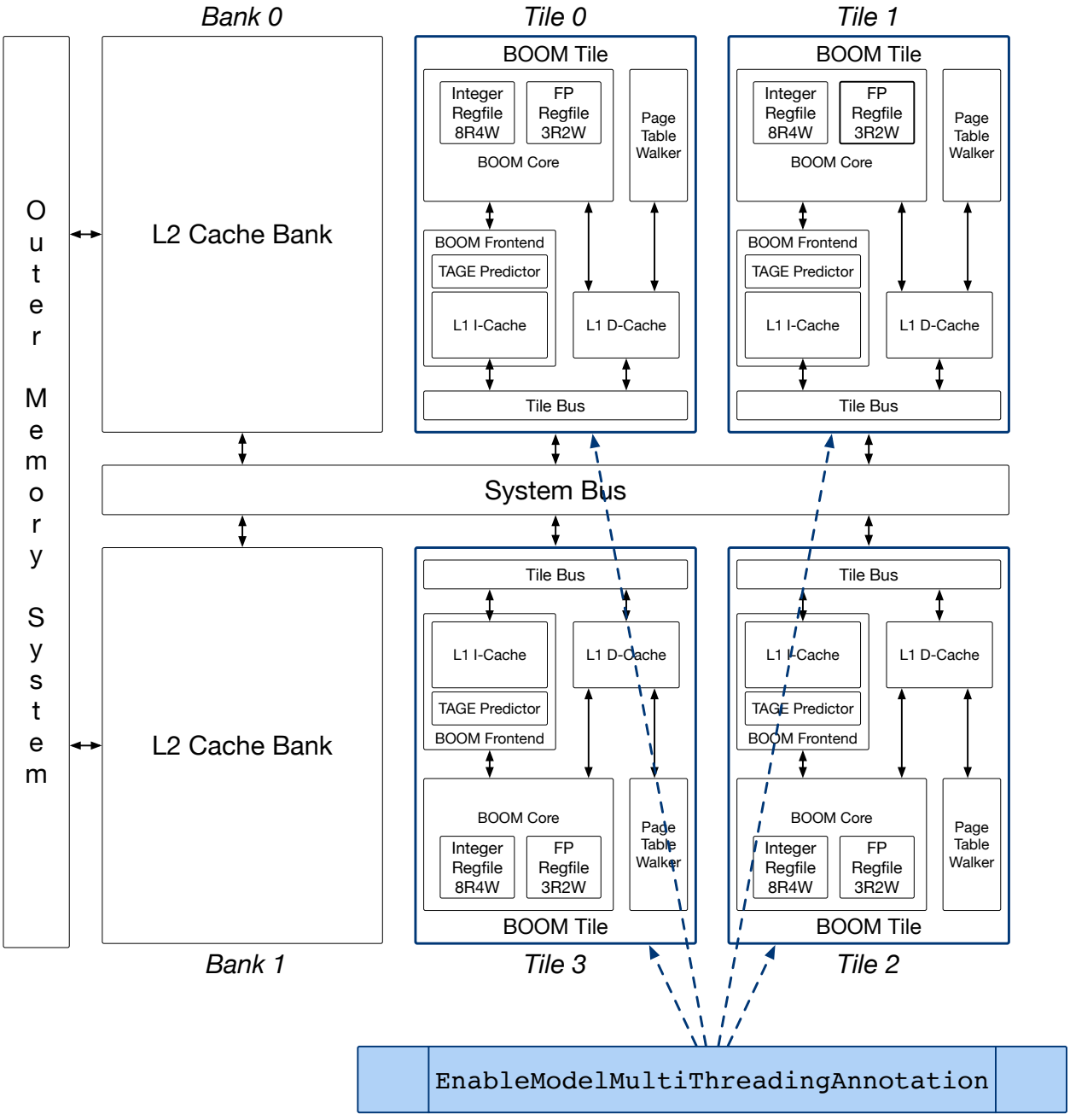

Figure 6.10: A hierarchical block diagram of a quad-core, LargeBOOM-based Rocket Chip system with associated annotations enabling instance multi-threading for simulation. Rather than threading individual cores, the tile containing each core and its associated caches is annotated as a member of the candidate set of identical instances for optimization.

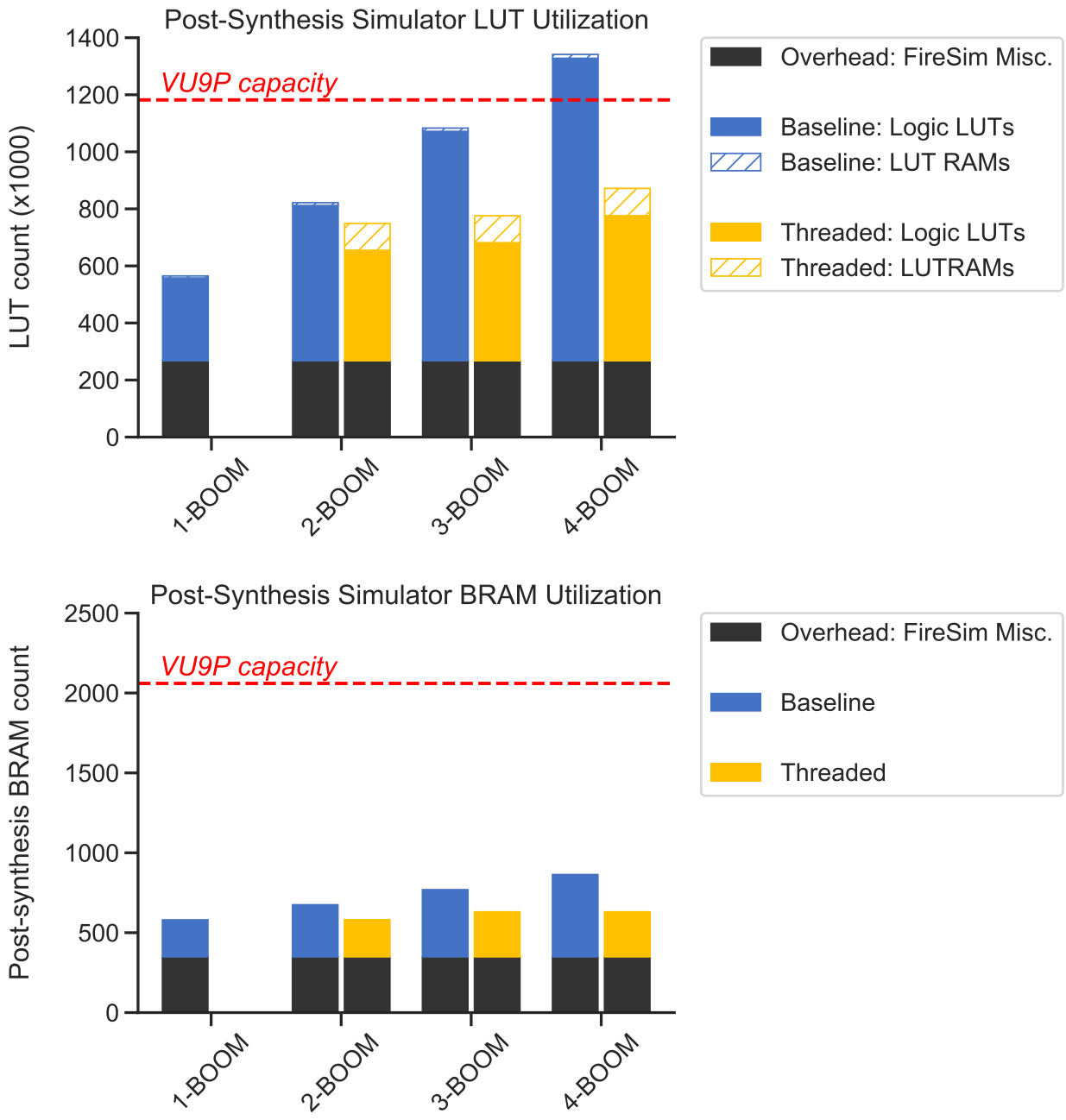

Figure 6.11: A comparison of FPGA resource utilization across baseline and threaded simulators of multi-core LargeBOOM-based Rocket Chip targets. Since the threading optimization requires multiple instances to target, only the baseline utilization is reported for the single-core system. Each simulator was generated with Golden Gate and synthesized with Vivado 2018.3. Here, we report post-synthesis utilization values that illustrate relative utilization of configurations that cannot successfully route and close timing for a VU9P device. While the two- and three-core threaded implementations closed timing at 50MHz, and the four-core threaded configuration closed timing at 45MHz, the unoptimized simulators of three- and four-core systems failed during implementation.

number of LUTs used as logic, the state-threading mechanisms described in Sections 6.3.5 and 6.3.5 offset this benefit to a small degree by generating large numbers of small memories implemented as LUTRAMs.

In addition to LUTs, Figure 6.11 also compares the number of BRAMs used by each simulator. Despite the fact that a threaded model requires the same number of synchronous-read memory bits as the sum of all the instances it replaces, threading also yields significant savings in this metric. As discussed in Section 6.3.5, target designs often contain memories with depths too shallow to map efficiently to the fixed-size BRAM macros in modern FPGAs. Since the threaded model replaces sets of memories with deeper memories aggregating the contents of the full set, the optimized simulator may avoid significant vertical waste present in the baseline design.

While these synthesis results provide insight into utilization trends, the ultimate practical consideration for the simulation user is whether a given system actually fits on a given FPGA host platform. To first order, this can be related to the number of resources of each type available on the device; however, the need to reserve significant numbers of resources to implement interconnect during place-and-route renders 100% resource utilization impossible in practice [28]. Therefore, Figure 6.11 depicts not only the the capacity of the Xilinx VU9P devices used by FireSim host platforms, but also an 85% utilization threshold for LUTs that is a reasonable heuristic for maximum achievable utilization for a large target with significant interconnect [85]. Indeed, this rule of thumb does correctly predict the outcome for our experiments, as the unoptimized simulators of both the three- and four-core systems failed during implementation.

While the baseline FireSim configuration could only simulate a maximum of two Large-BOOM cores, the optimized simulator is able to double this capacity to four cores. This does come at a performance tradeoff: as shown in Table 6.2, while the baseline simulators of one- and two-core systems execute at an overall FMR of 1.62, the two-core threaded simulator delivers only 37% of the throughput at FMR = 4.41. Clearly, given that the simulator would fit on a VU9P device in both cases, it would not be wise to employ threading for this two-core target. However, for a user simulating three- or four-core LargeBOOM systems, the performance of threaded simulator must be weighed against the need to either purchase a new host system or fall back upon a slower simulation modality. Indeed, the observed FMRs for three- and four-core simulators are 5.41 and 6.35 (at a reduced 45MHz host clock for the four-core configuration), which translate to an effective target clock cycle simulation rate of 9.24MHz and 7.07MHz, respectively.

When compared to other alternatives, these results represent relatively high throughput for simulating a large SoC with multiple 5-issue out-of-order cores. Though software RTL simulators offer ease-of-use and arbitrarily large capacity, their throughputs for multi-core SoCs are generally under 1kHz [2]. While partitioned simulators using multiple FPGAs can offer both scalable capacity and high speed, such systems often deliver only 1-10MHz of throughput [76, 2] at the cost of a high degree of complexity and designer effort [38]. In contrast, the threaded simulator offers performance comparable to a partitioned host using a cost-effective single-FPGA system and a user-friendly, automated compiler flow. By doubling

the capacity of a simple, cost-effective simulator while maintaining competitive speed, this optimization enables more effective full-system simulation of multi-core SoCs.

FMR	*N* LargeBOOM Cores			
	1	2	3	4
Unoptimized	1.62	1.62	⊗	⊗
Multi-threading	N/A	4.41	5.41	6.37†

Throughput	*N* LargeBOOM Cores			
Sim. cyc/sec	1	2	3	4
Unoptimized	30.9	30.9	⊗	⊗
Multi-threading	N/A	11.3	9.24	7.07†

Table 6.2: A performance comparison of baseline and threaded simulators. For each simulator configuration that could successfully be implemented on a Xilinx VU9P, observed FMR and effective throughput values were collected by booting into Linux and running a Python-based sorting benchmark. Both the 3- and 4-core LargeBOOM configurations failed to successfully route when the threading optimization was disabled. All feasible simulators closed timing at a 50MHz host FPGA clock frequency, aside from the four-core threaded configuration, where the † indicates a 45MHz max frequency. The ⊗ symbol indicates implementation failure during placement.

6.5.3 Broader Applicability to Accelerator-Based Systems

While the results with multi-core BOOM-based systems demonstrate the significant potential of instance multi-threading to optimize FPGA resource utilization, a key strength of this technique lies in its generality. Repeated instantiation of a common block is an extremely common design pattern across digital systems that helps exploit parallelism in a given application domain. Furthermore, the implementation of instance multi-threading as an automatic compiler optimization in Golden Gate allows this technique to be applied to simulators of arbitrary FIRRTL circuits to thread any set of identical instances. Therefore, to help illustrate this broader applicability, the experiments from Section 6.5.2 are repeated for a multi-core *domain-specific accelerator* based on the Gemmini systolic array co-processor for machine-learning applications [33].

Gemmini is a parameterizable co-processor generator that provides native integration with Rocket Chip and the Chipyard framework. It covers a design space of systolic array co-processors that arrange a number of small *processing elements* (PEs) in a two-dimensional mesh-connected network. By pairing this array with an assortment of boundary logic that includes a scheduler and a DMA engine, this array of PEs can perform efficient matrix-matrix

operations on data held either in a local scratchpad or the outer layers of the cache hierarchy of the enclosing SoC. By specifying different sizes and configuration options, Gemmini co-processors can capture the requirements of various matrix-multiplication kernels that are commonly found in the inference passes of deep neural networks (DNNs). Notably, Gemmini can be parameterized to support various number formats, including bfloat16 16-bit floating point values. This flexibility supports an agile design process where neural architectures and co-processor implementations can be rapidly co-designed to deliver efficient end-to-end performance.

As shown in Figure 6.12, a Gemmini co-processor can be paired with a general-purpose processor to form a complete accelerator core. The integration with Rocket Chip and Chipyard allows these accelerators to be composed with other hardware to create parameterized systems from one to many cores, with multi-level caches, multiple accelerators, and Linux-capable memory virtualization schemes. As with other domain-specific systems for machine learning, Gemmini-based chips may exploit various forms of data- or task-level parallelism through the use of multiple accelerator cores. Therefore, for this experiment, we consider a class of target machines that have varying numbers of identical Gemmini accelerator tile instances, each generated with the parameters described in Table 6.3.

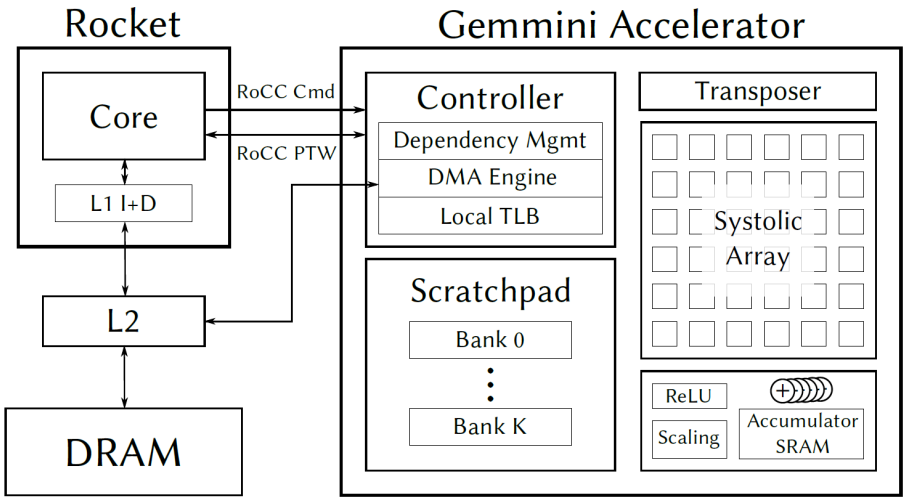

Figure 6.12: *(Figure used with permission from Hasan Genc [32])* A Gemmini accelerator pairs a general-purpose, scalar control processor with a systolic array co-processor for compute-intensive linear algebra operations. As with other systolic array-based accelerators, Gemmini is aimed at energy-efficient inference for machine-learning applications.

As Gemmini systolic array contains large amounts of logic, these "multi-Gemmini" systems can be extremely challenging to simulate with conventional FPGA compilation techniques. However, they present the natural opportunity to apply instance multi-threading in much the same fashion as with the BOOM-based systems: a *tile* including the accelerator,

Gemmini Accelerator Parameters	
Control processor	Medium Rocket (RV64GC)
Systolic array shape	Flat 8x8 grid, mesh network
Processing element	bfloat16 FMA & accumulator

Table 6.3: A high-level overview of the configuration parameters used to generate each Gemmini accelerator in the target designs for the instance-threading experiments. In a multi-core system, each of N tiles will pair such an accelerator with private caches.

private caches, and system bus integration is annotated with a Golden Gate compiler directive, and the set of all tile instances is simulated by a single threaded model. While this flow is effectively identical to that depicted in Figure 6.10, each tile now contains a significantly different internal implementation due to the presence of the large co-processor block and the use of a smaller Medium Rocket core with no hardware floating-point unit.

Figure 6.13 compares the overall number of resources required to synthesize the optimized and unoptimized simulators of systems with varying numbers of Gemmini accelerator cores. In contrast with the general-purpose BOOM designs, mapping the large number of floating-point fused multiply-add (FMA) blocks in the bfloat16 systolic array results in extensive use of DSP blocks on Xilinx FPGAs. Therefore, Figure 6.13 compares DSP block utilization along side that of LUTs and BRAMs, each as a fraction of the total number of each type of resource available on the host Xilinx VU9P FPGA. As with the BOOM-based target systems, instance multi-threading drastically reduces the number of logic resources–both LUTs and DSP blocks–used to simulate a given multi-core system. However, this effect is even more significant when simulating the multi-accelerator Gemmini systems: while an un-optimized FPGA compilation flow allows only a single accelerator core to fit on the VU9P host device, enabling optimizations makes it possible to simulate a much larger eight-accelerator design. While threaded simulation provides minimal relative reduction in memory implementation footprint, resulting in 56% BRAM utilization for 8 Gemmini cores, it is notable that the simulator of the 8-core design uses only 8.7% more LUTs and no more DSPs than the baseline single-core simulator. This highly efficient capacity scaling underscores the ability of the multi-threading optimization to fundamentally improve simulation capacity for general target designs that feature large, repeated instances.

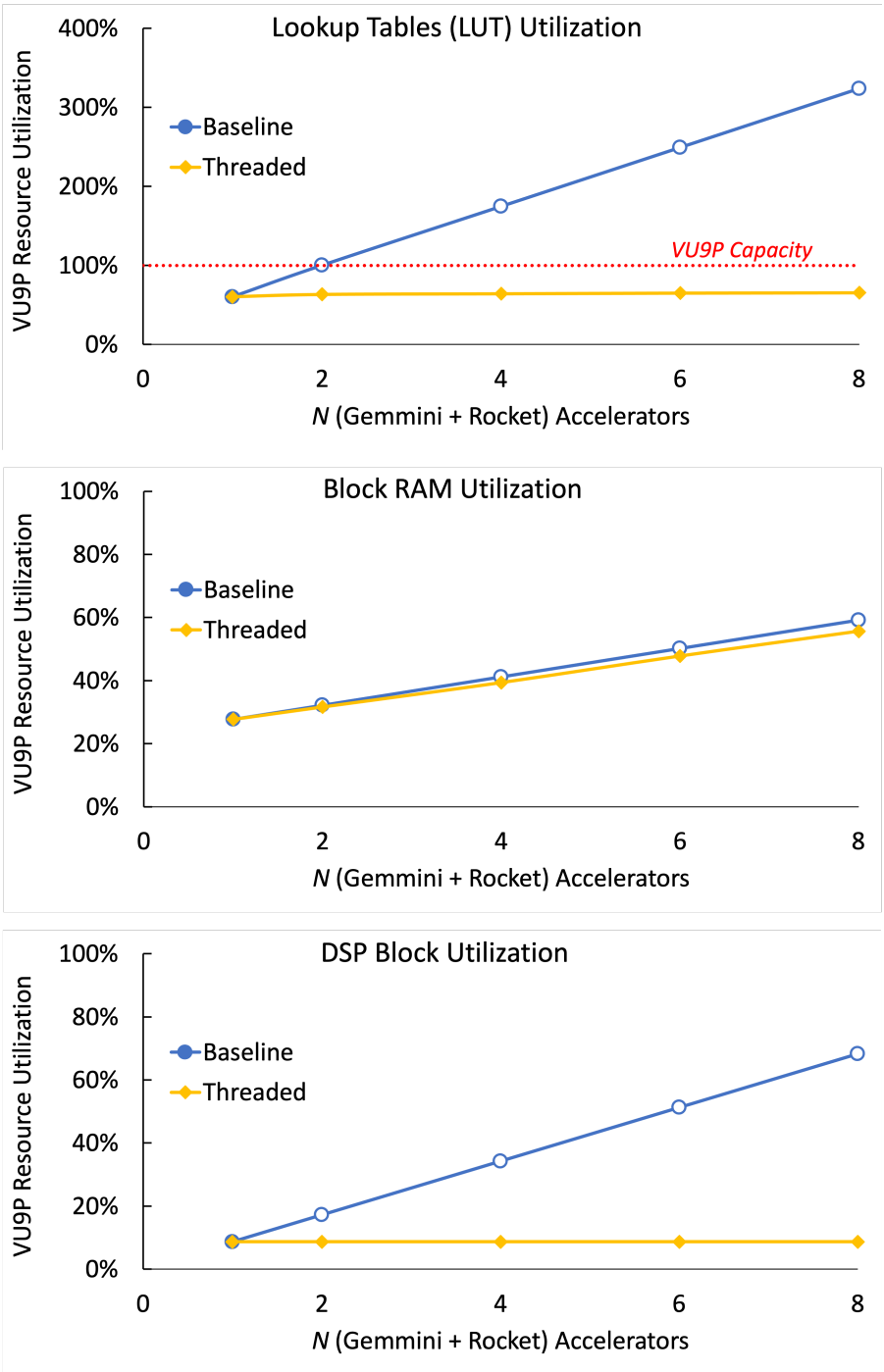

Figure 6.13: A comparison of FPGA resource utilization across baseline and threaded simulators of multi-core Gemmini accelerator targets. Hollow markers indicate post-synthesis utilization values for simulators that ultimately failed to map to a Xilinx VU9P FPGA, while solid markers indicate final utilization values for feasible simulators that completed implementation and closed timing at 45MHz. Here, the use of threading extends the simulation capacity of the VU9P FPGA from one Gemmini core to eight.

Chapter 7

Composing Multiple Resource Optimizations

As described in Chapter 3, Golden Gate is designed from the outset to support *hybrid simulators* that may combine multiple different implementation techniques. In this work, we describe how this architecture may support the targeted optimization of portions of the simulator that implement specific partitions of the target design. The optimizations described in Chapters 5 and 6 reduce the FPGA resource utilization of multi-ported memories and repeated instances, respectively, by generating multi-cycle implementations that effectively trade off simulation capacity and throughput. Each transformation generates a legal Latency-Insensitive Bounded Dataflow Network (LI-BDN) implementation of the corresponding partition of the target design. While the preceding chapters consider each optimization individually, demonstrating how the optimized component may be composed with a baseline LI-BDN model of the remainder of the target, a key advantage of this approach is the ability to compose multiple, independently optimized components.

With this capability to combine optimizations, we may explore a new region of the simulator design space employing both instance multi-threading and memory port serialization. With a closer eye on the tradeoffs of each individual optimization, we also explore their complementary nature; not only does each provide a benefit for distinct microarchitectural patterns found in target systems, but the use of threading provides the ability to "hide" the latency associated with other serializing optimizations, including that for multi-ported memories. Critically, in order to exploit the complementary benefits of the two optimizations, the simulator compiler must be capable of extracting fine-grained partitions of the target design; in practice, this might include complex topologies, such as a multi-ported memory that is contained hierarchically within a repeated instance that is to be threaded. Through a set of core transformations, the Golden Gate compiler framework establishes a mechanism for converting such topologies into a simple composite LI-BDN that is amenable to optimization.

With this groundwork in place, we outline a simple approach for combining the two optimizations within a single FireSim simulator. Since modern SoCs contain both repeated,

large instances (i.e., multiple cores) and expensive, multi-ported memories (i.e., register files), the basic tradeoffs of the two optimizations yield a simple template for greater resource savings than either optimization in isolation. Finally, we examine the performance of this strategy when applied to various configurations of the Rocket Chip SoC generator. While the relative performance varies across target designs, the combination allows a single FPGA to simulate previously intractable target designs, including a 16-core design using large BOOM cores, representing a 700% increase in simulated core count.

7.1 Combining Complementary Optimizations

Given that the ultimate goal of both optimizations is to reduce the number of FPGA resources required to simulate a given target design, a successful combination of optimizations should yield even greater resource savings than employing either alone. In the common case where a designer has access to FPGA devices of a fixed and finite capacity, this reduction in footprint will result in a corresponding increase in the maximum size of the target machine, allowing a simulator incorporating both multi-ported memory and multi-threading optimizations to simulate, say, a previously impractically large number of a particular processor core. However, since Golden Gate will generate a hybrid simulator where the optimizations will act on independent partitions of the target system, any further benefit of adding additional optimizations will be weighed against the benefit of simply applying one to a greater portion of the simulator. For example, if a design contains many repeated instances of a processor core that are prime targets for threaded simulation, applying the multi-ported memory optimization to memories in those cores reduces the scope—and potentially the benefit—of threading.

Fortunately, the two optimizations presented in this work have multiple complementary properties that suggest potential benefits in combination. Indeed, each is limited in applicability: the multi-ported memory optimization is only applicable to FPGA-hostile, highly ported memories, while threading can only be applied to a set of identical instances. In the hypothetical target design shown in Figure 7.1, an SoC might have a multiple instances of identical specialized accelerators alongside a single out-of-order core that itself contains a prohibitively complex register files. In this case, the two optimizations can naturally be applied to distinct components in the hierarchy by compiling the accelerator instances to a threaded simulation model and the register file to a multi-cycle memory model.

While systems like the one in Figure 7.1 show a simple motivation for combining optimizations, it is also worth considering the finer-grained, details of how each translates a target-design component to an optimized simulator. In cases where a repeated set of instances each contains a highly ported memory, both optimizations could theoretically be used—but which should? Here too, the optimizations are complementary: while threading reduces the footprint of the combinational logic by sharing it among simulated instances, it does not generally reduce the footprint of the state elements. As outlined in Section 6.3.5, though there are some marginal benefits in aggregating synchronous-read memories across

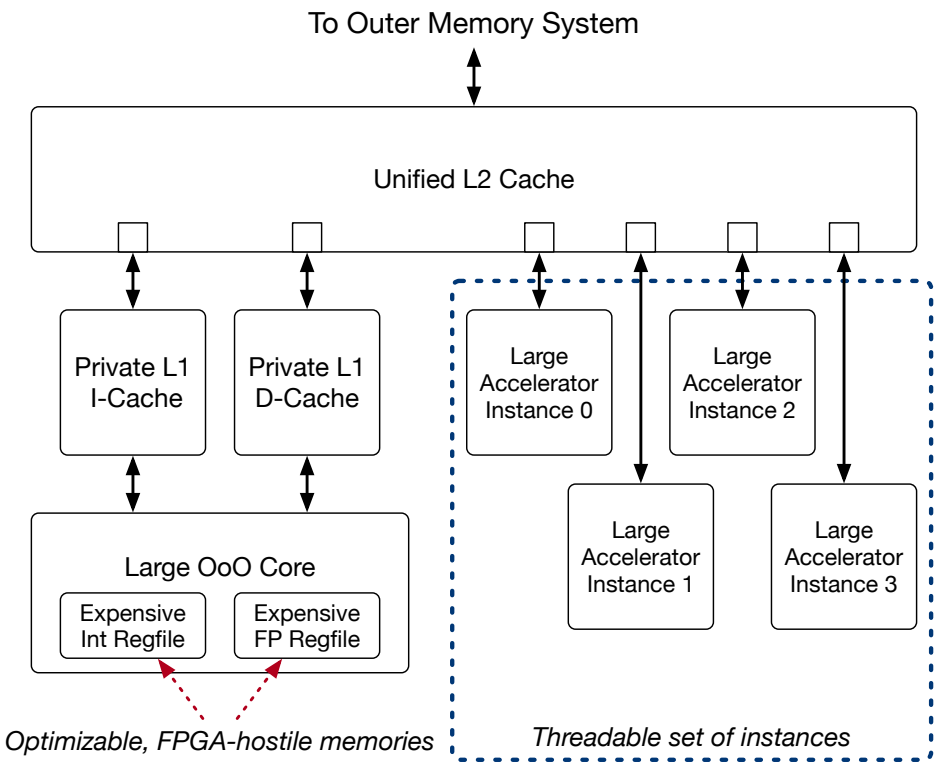

Figure 7.1: A realistic SoC that presents opportunities to save FPGA simulation resources by applying multiple optimizations, each to a distinct partition of the target hierarchy. The large, out-of-order core has two register files, each of which might be hostile to FPGA simulation due to high port count. Furthermore, the multiple instances of the same accelerator block present a natural opportunity for applying the multi-threading optimization.

instances, the logic to select a given thread's state increases the absolute cost of each register. Worse yet, threading has a pathological impact on highly-ported memories: since these FPGA-hostile memories generate vast multiplexing logic to implement their many ports, expanding their size to accommodate several copies has a super-linear logic cost. Given that this has the potential to significantly increase the absolute total cost of implementing the instances' memories, and given the reduction in cost of the instances' combination logic in the threaded model, the relative impact of these expensive memories can grow prohibitively large. In contrast, the multi-cycle memory models provide the maximum benefit for these difficult memories.

Table 7.1 qualitatively summarizes the relative benefits of each optimization for these different microarchitectural features; in short, it is clear that there is an opportunity for synergy. However, realizing this opportunity might require mixing the optimizations at a fine granularity to efficiently model repeated instances containing memories that are prohibitively expensive in a threaded implementation. Here, we describe how the Golden Gate compiler framework can manage these complex target design topologies. Furthermore, we highlight

how this composition can benefit from efficient thread scheduling, helping to reduce the relative performance cost of adding an additional degree of serialization to the simulator.

	Logic	SRAMs	Register Files	Registers
Memory models	no effect	no effect	large savings	no effect
Multi-threading	large savings	negligible savings	large cost	moderate cost

Table 7.1: A qualitative comparison of the resource-saving potential of the two optimizations when transforming various features of the target system's microarchitecture. The complementary nature of the benefits suggests the potential for even greater resource savings in combination.

7.1.1 Transforming Target Design Topologies

While Figure 7.1 may depict an idealized scenario,e the desired targets for each of the two optimizations are not always removed from each other in the hierarchy. Indeed, as shown in Figure 7.2, it is common for a target system to have multiple instances of a logic-heavy module that themselves contain FPGA-hostile multi-ported memories within their individual submodule hierarchies. This topology requires careful consideration: not only will the inner memory and outer logic-heavy module each be simulated by a dedicated LI-BDN, but the primitive LI-BDN simulating the outer module will exclude the memory.

Fortunately, this simulator structure can easily be obtained by properly transforming the target. As discussed in Section 4.3.1, both the memories and the enclosing logic-heavy modules must be extracted to the top level during the canonicalization of the circuit. By topologically sorting the module hierarchy graph and then extracting instances in order from highest to lowest in the hierarchy, Golden Gate may obtain the desired top-level partitioning of the circuit. Furthermore, this ordering avoids the creation of spurious wires running up and down the hierarchy that would otherwise increase both latency and area.

7.1.2 Hiding Optimization Latency with Threading

In software systems, multi-threading helps maximize utilization of finite hardware resources by reducing the impact of long-latency operations. By interleaving units of work from different computational threads, the number of dependencies in a processing element's instruction stream can be reduced; as a result, when a memory access that misses in cache or another similarly costly event occurs, the system may continue scheduling useful work from other threads that by definition do not depend on its completion. This concept may be realized through a wide variety of concrete implementations and scheduling patterns: systems may interleave individual instructions on a single processor core or switch when block, and they may operate with either static or dynamic thread orderings. Though these approaches may

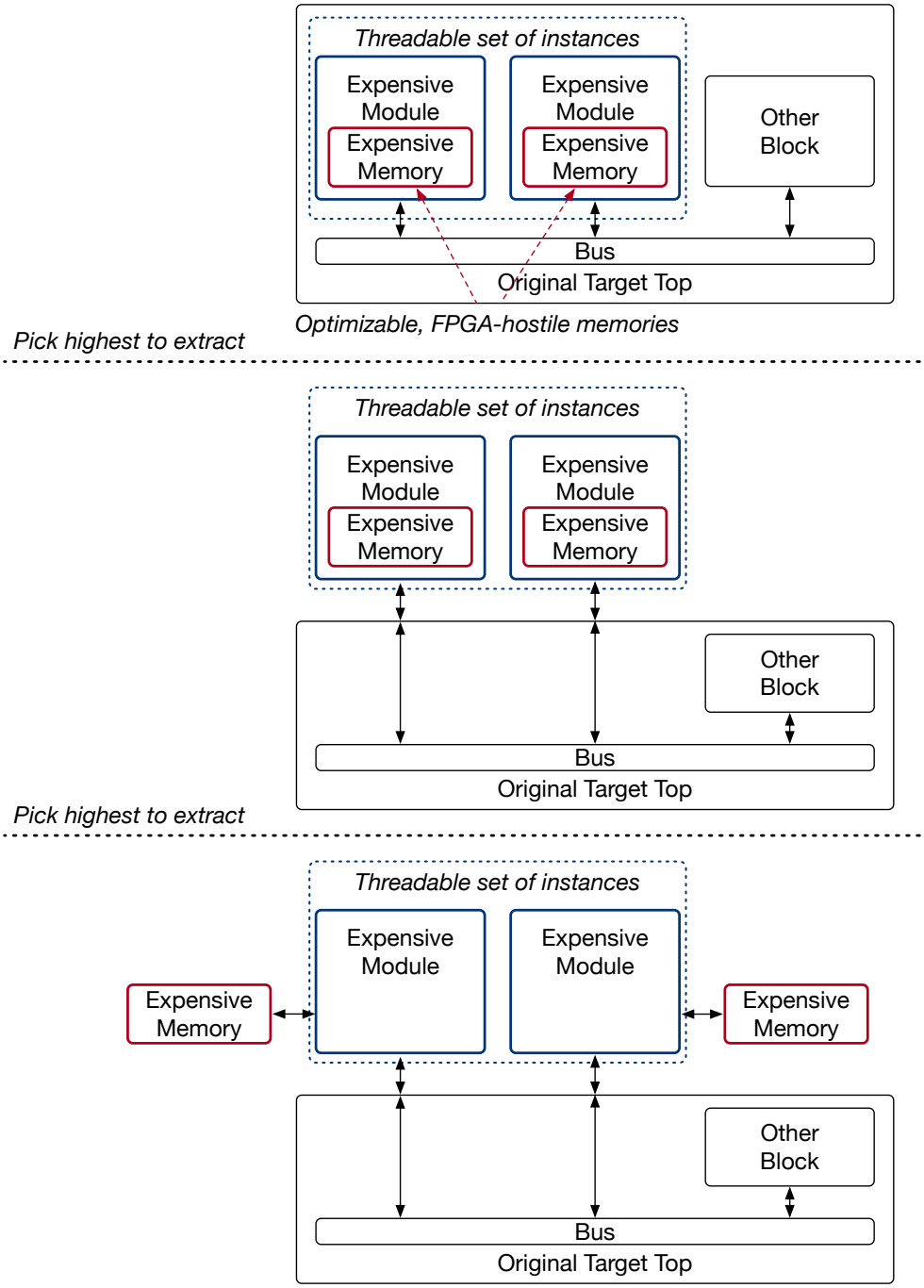

Figure 7.2: A depiction of the strategy used by Golden Gate to transform a target hierarchy with nested optimization targets. The instances in the target design are topologically sorted, and modules earlier in the order—which are "higher" in the the hierarchy—are extracted to the top level before proceeding to extract modules later in the order. This produces a canonical form with the fewest possible connections between the modules at the top level.

vary in ultimate performance, all provide a fundamental capability to increase throughput by *hiding latency* in the system.

While the use of the term "multi-threading" to describe the technique of scheduling the simulation of multiple instances onto one underlying model is common in previous work [71, 83, 84], the comparison is muddled by the atypical baseline for multi-threaded simulators: naive, fully parallel simulators. Indeed, in the common pattern of composing a threaded model for N cores with an unoptimized model for the remainder of the SoC, threading provides no latency-hiding benefit. While a given thread can dispatch tokens to the unoptimized portion in a decoupled manner, this "operation" takes only a single cycle; in other words, there is no extended latency to hide.

Fortunately, it is possible to take advantage of this latent latency-hiding capability by introducing other optimizations into the system. The serialization of accesses employed by the multi-ported memory optimization increases the latency to simulate a target cycle of a given memory to several host cycles. As discussed in Chapter 5, pairing these resource-optimized memory models with a baseline simulator can degrade performance, as the single-cycle baseline model is frequently waiting for the serialized memory operations to complete. In contrast, if the I/O tokens for the multi-cycle memory model are dispatched from a threaded model, the simulator is capable of maintaining its peak theoretical throughput of one simulated instance per host cycle, successfully hiding the latency of the memory operation. While this scenario represents an idealized case where all external I/O is ready each cycle, empirical results demonstrate that the thread scheduling is capable of mitigating the relative impact of other latency-increasing simulation techniques.

7.2 Evaluation

As in Chapters 5 and 6, our experimental evaluation compares both the FPGA resource utilization and performance of optimized and unoptimized simulators across a space of Rocket Chip-based target designs.

7.2.1 Applying Multiple Optimizations to Rocket Chip

A key motivation for the modular structure of Golden Gate simulators is to allow optimizations to independently target the components of the target design where they will have the greatest impact. Therefore, when mixing the instance threading and multi-ported memory optimizations, we may take advantage of the complementary nature of their benefits by directly combining the optimization strategies from Chapters 5 and 6. Since each optimization is enabled for a particular block in the target design by adding an Chisel annotation, this combination may simply add the union of both sets of annotations to the target.

Figure 7.3 depicts the same quad-core LargeBOOM-based Rocket Chip system as Figure 5.2, where it was annotated to replace each register file with a multi-cycle model, and Figure 6.10, where it was annotated to thread the set of BOOM tile instances.

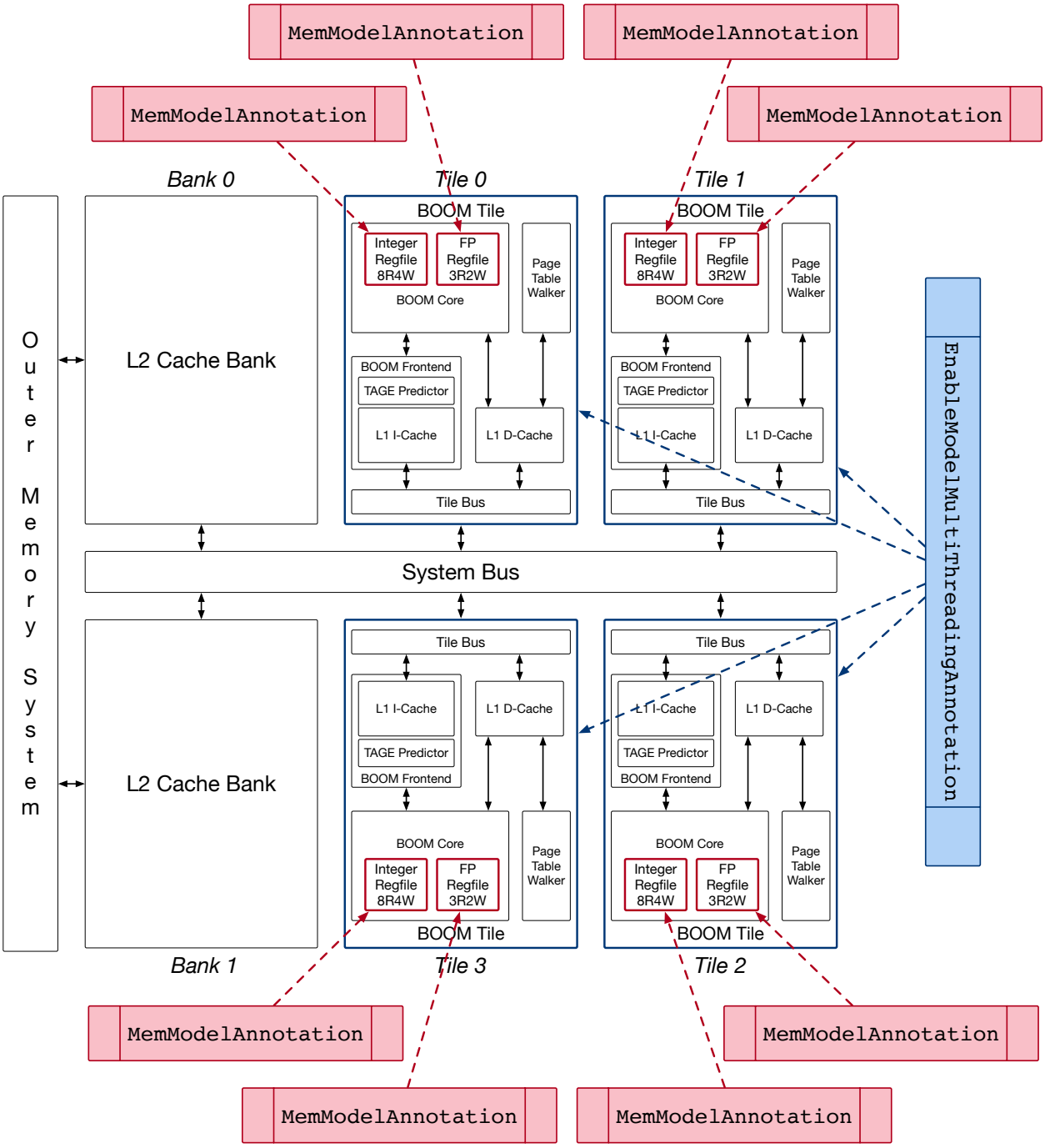

Figure 7.3: Applying multiple optimizations to Rocket Chip.

7.2.2 Experimental Results

As in Chapters 5 and 6, we evaluate the performance benefit of composing multi-ported memory serialization with instance multi-threading by measuring FPGA resource utilization for various simulator configurations. Using Rocket Chip targets with a varying number of LargeBOOM cores, these results may be compared among simulators of target designs of widely varying size. However, rather than simply comparing the results from the combined optimizations against those of the baseline simulator, we compare four different optimization strategies:

- No optimization
- Applying the multi-ported memory optimization to BOOM's register files
- Threading the tiles in an N-core system
- Combining both memory and threading optimizations in a *multi-optimized* simulator

Note: The register file optimizations use the dual-banked implementation from Section 5.4.3.

For each optimization strategy, simulators of target configurations with between 1 and 16 LargeBOOM cores were generated with Golden Gate and synthesized with Vivado 2018.3. As in previous sections, the ability to obtain synthesis results for simulators incapable of successfully completing place-and-route on the default FireSim host VU9P FPGA helps illustrate utilization trends across a wider range of core sizes. These results are depicted in Figure 7.4. In contrast with the more detailed results shown in Figures 5.3 and 6.11, we compare only overall LUT utilization, as block RAMs and other hard macros are never a tight constraint on simulator implementation.

Overall, the trends shown in Figure 7.4 demonstrate clear synergy between the two optimizations. The various *simulator configurations*—consisting of a target core count and an optimization strategy—displayed widely varying scaling trends. Both the unoptimized and memory-optimized simulators display nearly linear scaling of post-synthesis LUT count with core count, but with the memory-optimized simulator gaining on average only 70% as many additional LUTs per additional core when compared to the baseline simulator. In contrast, both the threaded and multi-optimized simulators demonstrate—predictably—far slower growth in LUT count with each added target core. Ultimately, the combined-optimization strategy displays the best scaling, in part by avoiding the paradoxical effect of FPGA-hostile memories growing super-linearly under threading. Indeed, the growth in LUT count is so small that the 16-core multi-optimized simulator requires only a 5.1% greater number than the 8-core configuration.

This sub-linear scaling provides significant resource savings at high core counts: though a 16-core simulator would normally require an infeasibly large 4.3 million LUTs, the combination of the two optimizations reduces this to 0.74 million, an 83% reduction. As the multi-ported memory serialization and instance threading themselves provide 28% and 67%

reduction, respectively, the combined strategy represents an almost ideal composition of their individual benefits.

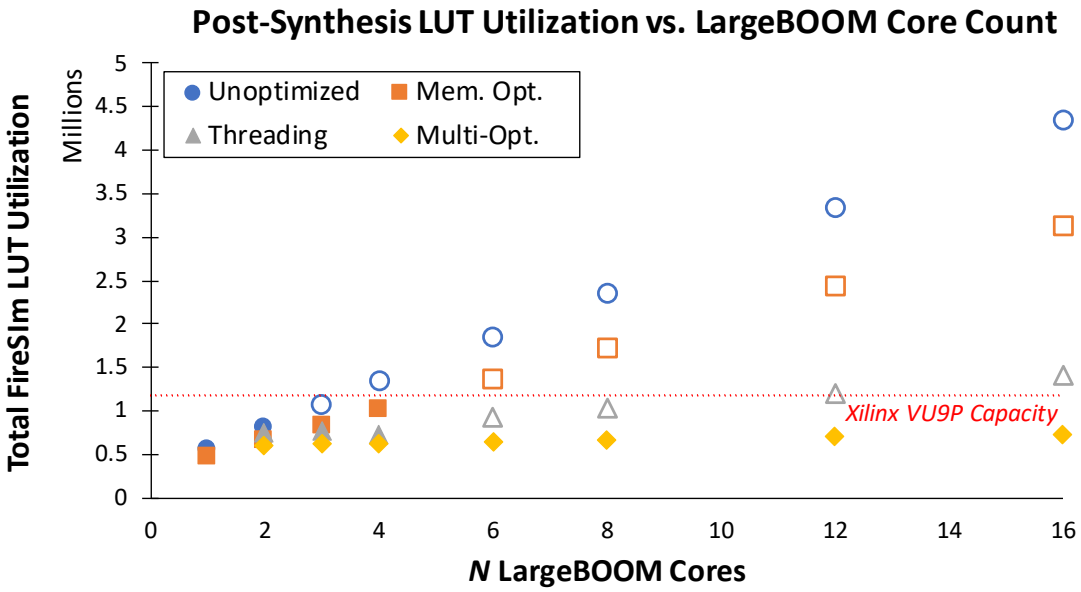

Figure 7.4: A comparison of the utilization of simulators modeling Rocket Chip SoCs with varying numbers of LargeBOOM cores. The simulators utilize four different optimization strategies: no optimizations, employing multi-cycle models to simulate the cores' eight-read/four-write register files, using one threaded model to simulate the set of core complex tiles, and using both optimizations in concert. The shaded markers represent simulator implementations that successfully routed and closed timing at a 45MHz or 50MHz host clock frequency, while the hollow markers represent post-synthesis utilization figures for infeasible simulators that failed during placement. Note that at least two cores must be present to employ the threading optimization.

While LUT utilization provides some insight into the scaling of the simulators across target machine size, it is ultimately only a proxy for the ultimate metric: whether the simulator successfully completes place-and-route and closes timing. Therefore, Figure 7.4 also depicts whether a given simulator configuration was successfully implemented for the Xilinx VU9P on used on the AWS F1 host instance. While a maximum of two LargeBOOM cores could be simulated with the baseline simulator, and each of the two individual optimizations could extend this to four, applying both optimizations allows sixteen cores to be simulated on the same VU9P device. This represents a 700% improvement in simulation capacity, radically extending the capability of the simulator. Here, feasible simulators closed timing at a 50MHz host FPGA clock frequency using Vivado 2018.3 and the default routing strategy prioritizing timing closure, aside from the four-core threaded configuration, where the † indicates a 45MHz max frequency. Failing designs were re-attempted with a congestion-optimizing

routing strategy, but this did not result in success for any previously infeasible simulator configuration.

FMR	N LargeBOOM Cores							
	1	2	3	4	6	8	12	16
Unoptimized	1.62	1.62	⊗	⊗	⊗	⊗	⊗	⊗
Regfile models	5.37	5.40	5.40	5.39	⊗	⊗	⊗	⊗
Multi-threading	N/A	4.41	5.41	6.37†	⊗	⊗	⊗	⊗
Both optimizations	N/A	6.14	9.11	8.11	18.1	24.0	24.0	32.0

Table 7.2: Observed FMR values for simulators with varying core counts and optimization strategies. For each simulator configuration, an N-core LargeBOOM target system was mapped to a simulator using the indicated optimizations, and, if implementation and place-and-route succeeded, observed FMR data was collected by booting into Linux and running a Python sorting application. Here, the ⊗ symbol indicates implementation failure.

Finally, in order to determine the relative performance of each simulator, average FMR values were obtained for each feasible simulator configuration across a target workload consisting of booting Linux and running a Python sorting script. The results of these experiments are shown in Table 7.2. Each additional optimization trades parallelism for reduced utilization; therefore, the multi-optimized simulator displays much higher FMR than the baseline, with a 16-core simulator requiring 32.0 host FPGA cycles per simulated cycle. However, while this is significantly slower than the 1.62 average FMR for the unoptimized simulator, these results must be interpreted in the proper context. Ultimately, the combined optimizations extend the capabilities of the simulator, not only avoiding a costly fallback to a far slower simulation methodology, but also allowing it to perform more useful work per simulated target cycle.

Comparing Simulator Capabilities

While the general utilization and throughput trends shed light on the performance of the individual and combined optimizations, the ultimate goal of this work is to help circumvent the restrictions previously imposed by limited FPGA hardware resources on simulation capacity. Therefore, it is worth comparing the simulation capacity supported by each optimization strategy for a reference FPGA; in this case, the Xilinx VU9P device found in AWS F1 instances. For a multi-core SoC like Rocket Chip, we define this capacity in terms of the maximum number of cores that may be simulated on the device with a given optimization strategy. Though this is ultimately an imperfect metric of simulation capacity, it reflects a valuable real-world use case, and a space of various core configurations may be used to reflect some of the variation in relative benefit of each optimization with changing target microarchitectures.

While the resource-saving optimizations do provide corresponding increases in simulation capacity, this comes at a notable increase in the FMR relative to an unoptimized simulator—fortunately, the potential host clock frequency is generally not slowed due to the FPGA-friendly microarchitectures generated by the optimizations. In the case where a user has access to a fixed set of FPGA resources that could not accommodate an unoptimized simulator, this may represent an acceptable tradeoff, as the alternative is to resort to far slower software RTL simulation; in other cases (e.g., interactive target-machine workloads), users may be extremely sensitive to simulator performance.

However, while any fair comparison of simulator capabilities must take these performance effects into account, FMR alone does not tell the complete story. A simulator modeling a larger system effectively does more useful work per simulated target cycle. Indeed, in a software simulator or commercial emulation system (see Section 2.1.2), it is generally expected that a larger design will result in a lower simulation throughput on the same host computer. Though this relationship is less direct in traditional FPGA simulation, the ability to trade off capacity and FMR with Golden Gate optimizations necessitates a performance metric that takes into account the size of the simulated target system. For the multi-core SoC target designs examined in this work, we contend that *simulated core-cycles per second* is a useful, size-aware metric for simulator throughput.

Table 7.3 summarizes the capabilities of the simulators employing the different optimization strategies in terms of both capacity and performance. Using the LargeBOOM configuration of the BOOM core, the maximum simulated core count increases from 2 cores with no optimizations to 16 cores with both multi-ported memory and multi-threading optimizations. While the ability to simulate a system with an 8x increase in core count comes at a significant 19.8x increase in FMR, it is notable that the number of simulated core-cycles per second is far closer, at 25.0MHz for the optimized simulator compared to 61.5MHz for the baseline. This is in spite of the fact that the optimized simulator uses only one replica of the combinational logic "datapath" to implement the state transition function for each instance, compared to the two copies in the unoptimized two-core simulator. Ultimately, this illustrates that the optimized simulator is relatively effective at keeping this datapath busy; while the baseline simulator must frequently stall to wait for external I/O or co-simulated models, the threaded scheduling is able to hide these latencies, along with the significant latency of the multi-cycle memory models.

Optimizations	Max Cores	FMR	$\frac{\text{simulated core-cycles}}{\text{FPGA cycle}}$	$\frac{\text{simulated core-cycles (millions)}}{\text{second}}$
Unoptimized	2	1.62	1.23	61.5
Regfile models	4	5.39	0.74	37.1
Multi-threading	4	6.7†	0.63	28.2†
Both optimizations	16	32.0	0.50	25.0

Table 7.3: A summary of the capacity and performance capabilities of simulators employing various combinations of optimizations for multi-core SoCs based on the LargeBOOM configuration of the BOOM core. Simulated core-cycles per second is a simulation throughput metric that incorporates the size of the target design.

Chapter 8

A Chisel Temporal Property Verification Toolkit

In the modern era of hardware design, the process of verifying a design against its specification represents a significant and growing burden. Broad consensus indicates that for many projects, verification accounts for a larger share of total engineering effort than design [31]. While the tools presented in this work are structured as compiler optimizations for generating FPGA-based simulators, they also represent a significant body of hardware design implementation artifacts; therefore, they cannot escape the burden of verification. Each new Golden Gate compiler pass is a form of hardware design generator, and some optimizations introduce parameterized instances of hand-written blocks to efficiently model components such as highly ported memories. Though FPGA-based simulators and associated software tools do not share the extreme risk of post-silicon bugs intrinsic to the ASIC design process, it is impossible for Golden Gate to provide a useful platform without significant efforts toward quality assurance.

Given this reality, the implementation of the resource-efficiency optimizations must incorporate verification collateral. While the limited resources of "gradware" development are a significant constraint, the use of the Chisel language and compiler ecosystem present both challenges and opportunities for functional verification. Though Chisel generally lacks the mature tool support of legacy hardware description languages like Verilog or VHDL, the flexibility of both the Scala frontend and the FIRRTL compiler framework allow users to build their own tools. With this in mind, we present two tools developed in the process of verifying elements of Golden Gate-based optimizations: a temporal property verification library, discussed in this chapter, and LIME, a specialized tool for checking simulation models, discussed in Chapter 9.

8.1 Background

Since its introduction in 2011, the Chisel language has cited as a driver of increased productivity for digital hardware designers across multiple application domains. Its focus on parameterization has naturally lent itself to developing generators that can produce variants of a basic design that span different sizes [90], and the use of Scala as a host language has allowed users to design new programming paradigms to extend reuse beyond the module level and to automate bus-level integration and parameterization [3, 26]. Beyond the general added expressiveness of of Scala meta-programming, this success can be partially attributed to the design of Chisel as a Hardware Construction Language (HCL), which eschews many of the complicated semantics of existing Hardware Description Languages (HDLs) in favor of a simple API that encodes synthesizable RTL designs by construction [6]; this simplification makes it easier to reason about—and therefore to reuse—existing portions of Chisel code. However, this restriction leads to a notable hurdle: as a strict hardware construction language, Chisel has limited support for directly expressing hardware verification constructs. While designers enjoy increased productivity and improved code quality, Design Verification (DV) engineers often rely on ad-hoc integration downstream of Chisel and FIRRTL; this gap is often cited as a challenge in reducing the relative burden of verification in a Chisel-based hardware development flow [67].

With the goal of helping to narrow this gap, we introduce a framework for incorporating Linear Temporal Logic (LTL) properties and Bounded Model Checking (BMC) into the verification of Chisel designs. This framework aims to bring the convenience of tight integration with Chisel while relying on the UCLID5 modeling system for performing the formal verification of the desired properties [11, 78]. By exposing an API at the Scala level, it aims to offer the reuse and productivity advantages of Scala meta-programming in the verification domain.

8.1.1 Related Work in Chisel Verification

In response to the perceived "Chisel verification gap," many existing projects have aimed to improve the verification flow for Chisel designs, especially in the domain of *dynamic verification*, where a Design Under Test (DUT) is simulated in the presence of particular input stimulus. The core Chisel API includes an `assert` operation that defines an *immediate assertion* requiring a particular Boolean-valued signal to be true at all points in time; this feature has proven to be valuable in encouraging designers to assume some of the verification burden. When combined with a basic convention for gating assertions before signals are reset, this has proven to be a mainstay of dynamic verification for Chisel designs. Furthermore, the ChiselTest framework provides a means to write low-overhead dynamic unit tests for hardware blocks by adding a simple peek- and poke-based API that runs within the same Scala programming environment as the generator of the DUT [66]. This simplified approach incentivizes designers to invest time in unit testing before delivering blocks to DV engineers, balancing out the traditionally back-loaded verification workload.

In addition to these core Chisel testing features, multiple research projects have aimed to better integrate formal methods for verification with Chisel. The SecChisel project introduced a set of tools for adding security labels to designs and checking the bounds of information flow [29], including a tool for generating SMT-LIB 2.0 [7] representations of FIRRTL circuits. While this tool is specialized for tracking the visibility of information and does not include full semantics for FIRRTL, it relied directly on FIRRTL IR without translation to Verilog, and prototyped a form of property propagation that predated the adoption of annotations in the FIRRTL implementation. Additionally, in the time since the development of the UCLID5 backend for FIRRTL, an experimental SMT backend for translating flat single-module FIRRTL designs directly to SMT-LIB 2.0 has been added to the FIRRTL repository [61].

In contrast to previous work on dynamic Chisel testing frameworks, the Chisel LTL toolkit provides tight integration with a model checker to enable low-overhead formal verification. Rather than requiring manually crafted or directed random stimuli reveal underlying bugs by elaborating large sets of traces, this approach allows the user to leverage the power of non-determinism to verify properties across the space of possible inputs. Furthermore, by building atop an existing model checker, UCLID5, via a thin abstraction, this open-source system provides the ability to accomplish useful tasks with less implementation overhead than other approaches to incorporating formal verification into a Chisel-based design cycle [21].

8.1.2 Linear Temporal Logic Properties

When defining properties for the LI-BDN networks that comprise a Golden Gate simulator, we rely Linear Temporal Logic (LTL) [74] formulae. LTL is a widely-used logic that enables the compact expression of specifications as temporal properties that are expressive enough to capture many safety and liveness properties as compact formulae.

In this work, LTL properties consist basic propositional logic formulae, along with a set of temporal operators. This may be expressed as a set of production rules for LTL property Φ. Here, Ψ_n is used to refer to any arbitrary LTL property, while p represents an arbitrary propositional logic formula. In the context of writing specifications for RTL designs, propositional logic formulae often derive their truth values from Boolean-typed signals, and a formula requiring such a signal to be true is effectively an immediate assertion.

8.2 UCLID5

The ultimate goal of providing an LTL property verification toolkit for Chisel is not to re-invent existing methods, but rather to provide productivity advantages through tight integration with Chisel and Scala. Therefore, in order to check LTL properties of Chisel systems, we rely on UCLID5, an open-source tool that defines a modeling language and provides an implementation that includes a variety of methods by which to verify proper-

Rule	Explanation
$\Phi = p$	Any propositional logic formula p is an LTL property
$\Phi = \neg\Psi_1$	Φ holds iff Ψ_1 does not hold
$\Phi = \Psi_1 \vee \Psi_2$	Φ holds iff Ψ_1 holds or Ψ_2 holds
$\Phi = \Psi_1 \wedge \Psi_2$	Φ holds iff Ψ_1 holds and Ψ_2 holds
$\Phi = \mathbf{X}\,\Psi_1$	Ψ_1 holds for the next time step
$\Phi = \mathbf{G}\,\Psi_1$	Ψ_1 holds for all traces starting at subsequent steps
$\Phi = \Psi_1\,\mathbf{U}\,\Psi_2$	There is some step where Ψ_2 holds, and Ψ_1 holds for all earlier steps

Table 8.1: A list of production rules summarizing the range of LTL properties used in this work.

ties of systems, along with other advanced tools for reasoning about or synthesizing their implementations [87].

At its core, the Chisel LTL property library relies on translating the FIRRTL circuits emitted by Chisel and their associated properties to a format that can be be programmatically verified. Furthermore, it is ideal that the format support a few key features:

1. Modeling of concurrent hardware systems and fine-grained bit-vector implementations
2. A module-level abstraction that reflects the organization of Chisel designs
3. One or more languages for expressing temporal properties

While there are a number of appropriate formats that provide all three of these features, including those of finite state model checkers like NuSMV [24, 23] and even SystemVerilog with its full assertion language [41], we choose to target the UCLID5 language for two primary reasons. Unlike SystemVerilog, which as of this writing requires commercial tools such as Cadence JasperGold [48] to check expressive temporal properties, there is a fully-featured open-source UCLID5 implementation. Furthermore, while FSM-oriented languages like NuSMV are sufficient to capture both Chisel circuits and associated LTL specifications, UCLID5 provides the capability to mix FSM-like and software-like components via modular abstractions. While this composition of models of both concurrent hardware and procedural software is outside the scope of this dissertation, there is active interest in modeling behaviors of hardware-software systems that reflect the realities of present Chisel implementations, including security properties of the Keystone open-source enclave [62].

While directly translating FIRRTL circuits provides both convenience and fidelity, the extremely low level of abstraction can also lead to high computational complexity for verification tasks. While RTL detail can be useful to reason about the true implementation of a particular block in a real SoC, it can lead to intractable SMT formulae. By reasoning about FIRRTL-level implementations only where necessary or convenient, the ability to blend com-

ponents automatically derived from real Chisel RTL with efficient, abstract software models can provide a useful tool in solving higher-level verification problems [19].

8.3 A Chisel-Based LTL Property Verification Flow

With these background goals established for the project, we may outline the concrete structure of the project. Just as the Chisel design flow includes a generator, frontend API calls, emission of FIRRTL, transpilation and lowering of FIRRTL, and translation to Verilog, the Chisel LTL property-checking toolchain involves multiple layers.

1. A user-facing Scala API for expressing LTL properties
2. Optional, custom FIRRTL transformations to encode verification-related assumptions
3. A backend for FIRRTL that encodes systems and properties in the UCLID5 language

In the core FIRRTL compiler, information is passed among layers via the well-specified FIRRTL IR. However, since this IR lacks both temporal properties and directives to control downstream verification tools (e.g. a BMC bound), the layers of the Chisel LTL toolkit must agree upon a standardized *verification IR* to represent such information.

In addition to the layered structure of the design flow, it is also intuitive to consider the user-facing verification language as a collection of layers, each providing different functionality. Here, we draw inspiration from the IEEE Property Specification Language (PSL) standard for cross-language temporal properties [39]. Though PSL includes a strictly more expressive language than LTL that is beyond the scope of this work, it also formally divides the property specification interface into four layers:

- The *Boolean layer* consists of Boolean expressions expressed via the host language; their truth values are defined at a particular instant in time.
- The *temporal layer* introduces temporal operators and may include Boolean-layer expressions as atomic propositions within the temporal property language.
- The *verification layer* contains standardized directives to control verification.
- The *modeling layer* uses the host language to define behavior of the system. When verifying an extant design, this includes the HDL implementation.

This layered structure is also a useful design principle for incorporating a temporal property library as a "third-party" addition to an existing HDL. In particular, we rely on Chisel and FIRRTL to provide the interface and the intermediate representation, respectively, for both the Boolean and modeling layers. However, since Chisel and FIRRTL semantics are insufficient to express the temporal and verification layers, these rely on a custom intermediate representation contained in FIRRTL annotations. In this implementation, the verification

IR is embedded within a collection of FIRRTL annotation types that rely on the annotation-propagation framework described in Section 4.2.2. These annotations broadly fall into two classes: property annotations, which encode the actual LTL specification, and control annotations.

8.3.1 LTL Property Annotations

When designing a means to encode LTL properties for Chisel designs, there are two critical objectives: the encoding must be sufficient to represent the full LTL property language from Table 8.1, and it must have a means by which to incorporate the value of signals in the current state of the system into the definitions of atomic propositions. Fortunately, since FIRRTL annotations can contain arbitrary serializable data structures and Scala case classes, it is possible to define annotation classes that carry Abstract Syntax Trees (ASTs) for arbitrary languages. Furthermore, annotations rely on the `Target` API introduced in Section 4.2.2 as a formalized interface for referring to signals within the circuit. By combining these two features, the annotations may encode arbitrary LTL properties that refer to atomic propositions that are defined at the Boolean layer as traditional components of the FIRRTL IR, while also properly tracking changes to these components that arise in the standard lowering passes of the FIRRTL compiler.

8.3.2 Control Annotations

In addition to a system and a set of temporal properties that encode its specification, a verification task generally requires a user to specify a strategy. One example of such a strategy is directing a model checker to perform BMC on a system against the specification; furthermore, this strategy may take parameters, such as the time bound in the BMC example. In general, these are elements of the verification layer from the PSL taxonomy.

8.3.3 Chisel LTL Property API

Given that the verification IR combines a very direct representation of LTL properties with FIRRTL annotation targets to track Boolean-layer signals, a very thin Scala frontend API is sufficient to implement the LTL language and control directives. By relying on Scala features aimed at embedding domain-specific languages,including symbolic method names, implicit conversions, and companion objects [6], the LTL property can be rendered as unobtrusive as possible.

As shown in Figure 8.1, these features allow properties to resemble traditional LTL notation. While the `AP` operator explicitly promotes Boolean-layer signals to the temporal layer, implicit conversions allow such calls to be omitted when passing arguments to higher-level LTL operators like F. Finally, wrapping the property in the `LTL` operator captures its syntax and associated references to Boolean-layer signals within a FIRRTL annotation. The

```
import chisel3._
import chiselucl.control._
import chiselucl.properties.ltlsyntax._

class MyModule extends Module {
  ...
  LTL(G(AP(active && start) ==> F(finish)))
  ...
```

Figure 8.1: An example liveness property expressed in the Chisel LTL language

net result is a convenient Scala frontend API that conceals the fact that the verification IR is an ad-hoc addition to the Chisel and FIRRTL ecosystem.

8.3.4 Verification Library Transforms

As a guiding principle, the FIRRTL infrastructure adopts a variant of the "Unix philosophy": it is ideal to divide tools into multiple transforms that each do one thing well. Therefore, since some steps of UCLID5-based LTL verification flow express fundamental patterns that are useful in a broader context, these components are packaged as independent, reusable transforms. Here, we examine two such transforms: a tool for transforming resets into initializations, and an alternative strategy for lowering arbitrary "don't care" values to better exploit non-determinism in verification.

Reset to Initialization Transform

When expressing systems in UCLID5 and other verification languages, first-class initialization constructs are used to abstractly represent both reset events and other initial-value assumptions that may exist for the system. In contrast, systems expressed in hardware description languages such as SystemVerilog that support both design and modeling tend to draw a dichotomy between *reset* and *initialization*. While the former defines how a signal assumes a known value in response to some stimulus, the latter defines the value the signal has at the "start of time." As this lacks a general physical analogue in an ASIC implementation of a digital circuit, the constraints of automated logic synthesis require designers to express resets via standard logic constructs such as conditional assignments. Explicit initialization features are often restricted to simulation contexts, where they takes precedence at the beginning of execution, or to the semantics of initial states when reasoning about the design as a transition system. However, some modern FPGAs define a precise "start of time" immediately after programming, at which point signals take their specified initial values that immediately follows programming.

Given the broad utility of initialization, it is a first-class construct in both UCLID5 and (System)Verilog. While the core FIRRTL specification lacks a first-class initialization

construct, support has been added to the default implementation via annotations: if an asynchronous reset signal is annotated as "preset" signal, all state elements that rely on that signal for reset will be initialized unconditionally with their initial value. Though more complex, this has a direct mapping onto dedicated initialization constructs in target languages (such as `initial` in Verilog) during emission and is therefore useful for simulation and verification purposes.

In order to better exploit this feature, the verification transform library includes the `ResetToInitialization` transform, which infers whether a design has a single reset domain; if it does, it converts all register resets to the initialization pattern. This presents a powerful simplification when generating downstream UCLID5 models, as it allows the initial state of the system to represent a consistent post-reset state. Furthermore, translating resets implemented with FIRRTL logic into UCLID5 `init` constructs significantly reduces the computational complexity of a model by eliminating the need to predicate checks on a preceding reset event. Though it is not universally applicable, this translation is convenient for checking properties in the common case of blocks with well-defined, simple reset semantics.

Arbitrary Value Substitution Transform

Like initialization, another common feature of both UCLID5 and SystemVerilog is the assignment of arbitrary value to a signal. In SystemVerilog, this is represented with the use of the `X` state as the assigned value (though the semantics of operations on this value render it difficult to use correctly [82]), while in UCLID5, the 'havoc' statement is used to perform a non-deterministic, arbitrary assignment to a variable. To enable multiple idioms for assigning and using arbitrary values, FIRRTL supports both an invalidation statement and a conditionally valid expression, and the compiler infrastructure includes transforms to help lower these constructs to support multiple target languages.

Although FIRRTL IR is offers multiple arbitrary value constructs, significant hurdles remain to productively using them for verification. In the present compiler implementation, the mandatory lowering of these constructs ultimately results in the removal of all arbitrary-value assignments in favor of deterministic assignment with zero; worse yet, disabling this transform will violate the assumptions of downstream transforms. Though this enhances simulation determinism, it eliminates the bug-finding power of non-determinism that is critical for techniques like BMC.

Given this limitation, and given the cross-environment appeal of selectively emitting true arbitrary value constructs, we include an `ArbitraryValueSubstitution` transform that relies on annotations and alternative, reversible lowering. By appropriately specifying their prerequisites, emitters that support arbitrary values in their target language may optionally select the alternative lowering, allowing appropriate representations of non-determinism to be carried through to the verification environment.

8.3.5 A UCLID5 Backend for FIRRTL

In contrast with the verification utility transforms, which are more generally applicable, the final layers of the Chisel LTL property-checking stack are highly specialized to the UCLID5 target language. However, many the benefits of a layered compiler still apply, even if the reusability is limited. Therefore, the UCLID5 backend is divided into two components: backend transformations and target language emission. In addition to the final emission layer, we highlight a Boolean type inference transform that enhances the quality of emitted models.

Boolean Type Inference

Though UCLID5 is generally well-matched for representing digital hardware, there are some semantic differences between it and FIRRTL that complicate emission. Most noticeably, since UCLID5 aims to provide a high degree of type safety, it distinguishes between single-bit bit-vector (or `bv1`) values and Boolean values. Though this mirrors VHDL [40] and provides a natural match for the intended SMT-LIB target format, it presents a significant mismatch with FIRRTL that unifies the two types. Most critically, the built-in LTL property component of the UCLID5 language *requires* atomic propositions to have Boolean type; therefore, some care is needed to ensure a consistent interface between the Boolean layer and the temporal layer in the emitted code.

To address this mismatch, the UCLID5 backend flow includes an additional transform to process Boolean expressions. Since UCLID5 lacks explicit type casts from `bv1` to Boolean or vice-versa, the ultimate fallback for type coercion is a comparison with unity or a ternary operator, respectively. However, excessive use of these patterns increases model size, which leads to increased solver runtime. Therefore, the Boolean type inference transform attempts to minimize the number of coercions using a min-cut algorithm on a typed dataflow graph.

UCLID5 Emission

At the end of the compiler flow, the lowered circuit must be translated to the UCLID5 target language. Fortunately, since this language is a good match for expressing basic hardware description languages like Low FIRRTL, and since the transforms described in the preceding sections further minimize any opportunities for semantic discrepancies, the process is fairly straightforward.

In the concrete Scala implementation, the UCLID5 emitter is generally similar in structure to the SystemVerilog emitter in the default FIRRTL flow. As with other FIRRTL backends, this emitter incorporates settings defined by certain annotations into its emission strategy, including the generation of `init` blocks based on the preset annotations introduced in Section 8.3.4. However, unlike other emitters, the UCLID5 emitter also processes property and control annotations and emits them as part of the output UCLID5 modules. By preserving the property annotations from the Chisel layer all the way through UCLID5 emission, the output LTL properties syntactically reflect the user input, enhancing readability.

8.4 Case Study: Verifying a Queue

As a concrete example of a task that could be solved with the Chisel LTL toolkit, we check that a simple Chisel implementation of a First-In, First-Out (FIFO) queue obeys a specification defined by a set of LTL properties. This problem mirrors the `Queue` example system from the UCLID5 repository and demonstrates how to add an LTL specification to a similar system specified in Chisel.

As a simplifying assumption, consider the basic one-entry `UnitQueue` shown in Figure 8.2. It has been augmented with several LTL properties that make up a specification, along with a BMC control annotation. While these properties will hold true on a deeper FIFO that uses an underlying memory to store enqueued elements, the `UnitQueue` reduces the code footprint of each example.

While the design is very simple, it demonstrates most of the key features of the Chisel LTL toolkit. Furthermore, it illustrates that while the Scala-embedded language makes it convenient to express a specification, devising an appropriate specification still requires careful consideration. Here, we consider three properties:

- The `output_irrevocable` property ensures that if the queue has an item available at its output that is not dequeued, the same item will be available in the next cycle.
- The `input_eventually_output` property ensures that any item that enters the queue will eventually be available at its output, as long as the downstream sink doesn't block the queue forever and there is no intervening reset.
- The `no_spurious_backpressure` property specifies that the queue cannot remain full (unable to enqueue) indefinitely if the downstream interface is attempting to dequeue.

Each of these properties mixes Boolean- and temporal-layer constructs. Additionally, the first two properties include constants – comparisons with constants of unspecified values can be used to "capture" values and use them as part of temporal specifications. This modeling construct is defined via the standard Chisel and FIRRTL blackbox interfaces by representing the constant with a wrapper of the appropriate data type. Finally, it is noteworthy that the latter two properties are *liveness properties* [70], which require infinite traces to exhaustively verify. This demonstrates one of the greatest advantages of relying upon an existing modeling system like UCLID5: it implements algorithms to translate bounded checks of liveness properties to SMT formulae that ensure the system does not re-enter a liveness-violating state within a particular *recurrence diameter* [9]. Through the combination of powerful tools with simple front-end interfaces, designers may rely on an ergonomic, fully automated flow to harness the power of non-determinism to check properties.

```
class UnitQueue(width: Int) extends MultiIOModule {
  val i = IO(Flipped(Decoupled(UInt(width.W))))
  val o = IO(Decoupled(UInt(width.W)))

  val data = Reg(UInt(width.W))
  val full = RegInit(false.B)

  i.ready := !full
  o.valid := full
  o.bits := data

  when (i.fire) { full := true.B }
  when (o.fire) { full := false.B }

  // If queue has data and it isn't dequeued, it still has that data.
  val arbitrary_valid_data = FreeConstant(UInt(width.W))
  LTL(
    G(AP(!reset.toBool && o.valid && !o.ready && o.bits ==
        arbitrary_valid_data) ==>
      X(o.valid && o.bits == arbitrary_valid_data)),
    "output_irrevocable"
  )

  // Data entering implies it eventually appears at the output,
  // unless the downstream sink blocks forever or reset occurs
  val arbitrary_input_data = FreeConstant((UInt(width.W))
  LTL(
    G(F(io.deq.ready)) ==>
    G(AP(i.fire && i.bits === arbitrary_input_data) ==>
      F(reset.toBool || (o.valid && o.bits === arbitrary_input_data))),
    "input_eventually_output"
  )

  LTL(
    G(AP(o.ready) ==> F(i.ready)),
    "no_spurious_backpressure"
  )

  // Run BMC for 10 steps: Queue-ED
  BMC(10)
}
```

Figure 8.2: A Chisel single-entry FIFO queue with an associated LTL specification.

8.5 Leveraging Generators & Object Orientation

As with the core Chisel hardware construction API, an added benefit of the Chisel LTL property library is the ability to embed API calls within a larger Scala program. In particular, object orientation can be a useful pattern for expressing a *verification generator*, and can enhance reuse of verification collateral by avoiding the need for repeated explicit inclusion in multiple components

While the properties in Figure 8.2 can all be used to verify the single-entry queue, it is possible to express some of them as more general properties of the interfaces of the queue. For example, the `output_irrevocable` property encodes irrevocability of the output ready-valid interface of the queue, and could broadly apply to many ready-valid interfaces that guarantee that a valid token will remain present on the interface with constant data until accepted by the sink. Indeed, the Chisel standard library has the notion of a ready-valid interface with such properties, with the `IrrevocableIO` class that is labeled with the following restrictions:

> A concrete subclass of ReadyValidIO that promises to not change the value of 'bits' after a cycle where 'valid' is high and 'ready' is low. Additionally, once 'valid' is raised it will never be lowered until after 'ready' has also been raised.

By associating this behavior with a Scala generic class, and therefore a parameterized type, Chisel is able to help provide a degree of safety from some errors. For example, one common error of composition would be to pass a source with a non-irrevocable `ReadyValidIO` output to a connection or function expecting irrevocable behavior; by encoding the distinction in the type system, generators can be structure to make this impossible. However, while strong typing is helpful, it is not a panacea, and the Chisel implementation notably lacks any integrated way to ensure that the requirements of the *IrrevocableIO* interface are actually honored.

Fortunately, it is possible to encode these requirements in an LTL specification to complement the benefits of type safety. Furthermore, these properties can be defined in Chisel LTL API calls contained within the definition of the interface itself. As shown in Figure 8.3, incorporating properties of an interface in the corresponding *Bundle* class definition allows them to be seamlessly applied to every instance. In this way, a library developer may leverage object-oriented design to enhance reuse of LTL properties and lower overall verification burden.

```
class IrrevocableWithSpec(width: Int) extends Bundle {
  val ready = Input(Bool())
  val valid = Output(Bool())
  val bits  = Output(UInt(width.W))

  val arbitrary_valid_data = FreeConstant(UInt(width.W))
  LTL(
    G(AP(valid && !ready && bits == arbitrary_valid_data) ==>
      X(valid && bits == arbitrary_valid_data)),
    "irrevocable"
  )
}


class UnitQueue(width: Int) extends MultiIOModule {
  val i = IO(Flipped(Decoupled(UInt(width.W))))

  // Irrevocable LTL property automatically applied to output!
  val o = IO(IrrevocableWithSpec(width.W))

  val data = Reg(UInt(width.W))
  val full = RegInit(false.B)

  ...

  // Run BMC for 10 steps
  BMC(10)
}
```

Figure 8.3: Similar to a Chisel generator, a *verification generator* can leverage object-orientation to ease reuse of common LTL properties.

Chapter 9

LIME: Verifying Multi-Cycle Models

With Golden Gate, we demonstrate that pervasive area optimization can be applied to a large, FPGA-accelerated hardware simulator by substituting FPGA-hostile elements of the design for FPGA-friendly models. In the construction of a decoupled, compositional hardware simulator, this substitution can include changes to the underlying latencies of isolated partitions of the design, allowing techniques like time-multiplexing of hardware resources to be applied without the typical limitations imposed by synchronous FPGA prototypes. However, these optimizations can only be practical if it is possible to establish with a high degree of confidence that the decoupled, optimized models of the design avoid breaking the cycle- and bit-exact correspondence of the simulation results with the ordinal target design. Fortunately, the LI-BDN formalism provides a framework in which this high-level correctness goal can be achieved; however, it in turn depends on the correctness of the individual models of the simulator for both cycle-accuracy and deadlock-avoidance guarantees.

While Vijayaraghavan and Arvind [89] describe the correctness conditions for individual simulation models as a set of formal properties, it can be difficult to check whether such an optimized model is truly "simulation equivalent," as the notion of equivalence is distinct from the trace containment concept used in other hardware equivalence checks. To address this gap, we introduce lime, a push-button model checking tool for checking the correctness of Golden Gate simulation models.

9.1 Structure of the LIME Checker

At a high level, verifying LI-BDN simulator implementations involves checking the three properties introduced in Section 3: Partial Implementation (PI) of the reference design, along with the No Extraneous Dependencies (NED) and Self-Cleaning (SC) properties that guarantee that the simulator will not deadlock. lime achieves this by automatically generating a Bounded Model Checking (BMC) problem for each of the three properties for a given model, with each BMC case structured as an input to the UCLID5 [78] verification system. The lime flow is depicted in Figure 9.1, which shows the how a model-checking problem

is created from FIRRTL circuits specifying the model (`Model.fir`) and associated target component (`RTL.fir`). LIME has two primary phases: it translates the FIRRTL inputs into the semantics of UCLID5, and then it constructs a model-checking problem for each of the LI-BDN properties.

9.1.1 A UCLID5 Backend for FIRRTL

To check formal properties of FIRRTL circuits like PI, NED, and SC, it is necessary to have both a formal model of FIRRTL semantics and an automated tool for representing FIRRTL circuits in a model-checking environment. To this end, we developed a UCLID5 backend for the FIRRTL compiler. As described in [47], the FIRRTL compiler is composed of many lowering passes that progressively remove higher-level constructs from the IR until it is in a lowered form. At this point, one of multiple *emitters* is invoked to produce output in a preferred form. Typically, designs are emitted as Verilog for use by downstream CAD tools, whereas LIME uses our UCLID5 emitter.

We targeted the UCLID5 modeling system as it is open-source, and provides strong support for compositional modeling across both synchronous and asynchronous systems. While LIME was designed to check LI-BDN properties, the UCLID5 backend is considerably more versatile. Using Chisel and FIRRTL, designers can write annotations that carry UCLID5 assumptions, invariants, and properties to be emitted alongside the UCLID5 implementation of the circuit. This enables designers of hardware generators to co-generate verification collateral, easing the challenge of verifying a generator with a large space of possible output designs. Since LIME is intended to help hardware designers write formally verified LI-BDN models, we extended UCLID5 to optionally emit VCD waveforms in order to make counterexamples easier to interpret.

9.1.2 Modeling Environment Generation

The *Environment Generator* is a Python program that generates UCLID5 testbenches to verify Partial Implementation, NED, and SC for a given reference RTL and LI-BDN model pair. Since each of these has slightly different modeling environment requirements, we split checking these properties into three separate testbenches. To enable a "push-button" verification tool, we use metadata produced during FIRRTL compilation to establish correspondences among the token channels of the LI-BDN simulation model and the I/O of the reference RTL component, and LIME automatically specializes the generated environment to have the appropriate structure. Because appropriate invariants for k-induction must constrain internal state of the simulation model, they require introspecting on the model implementation in a manner that is currently incompatible with our automatic testbench generation. Instead, we use Bounded Model Checking to verify the three properties, and leave an inductive approach to future work.

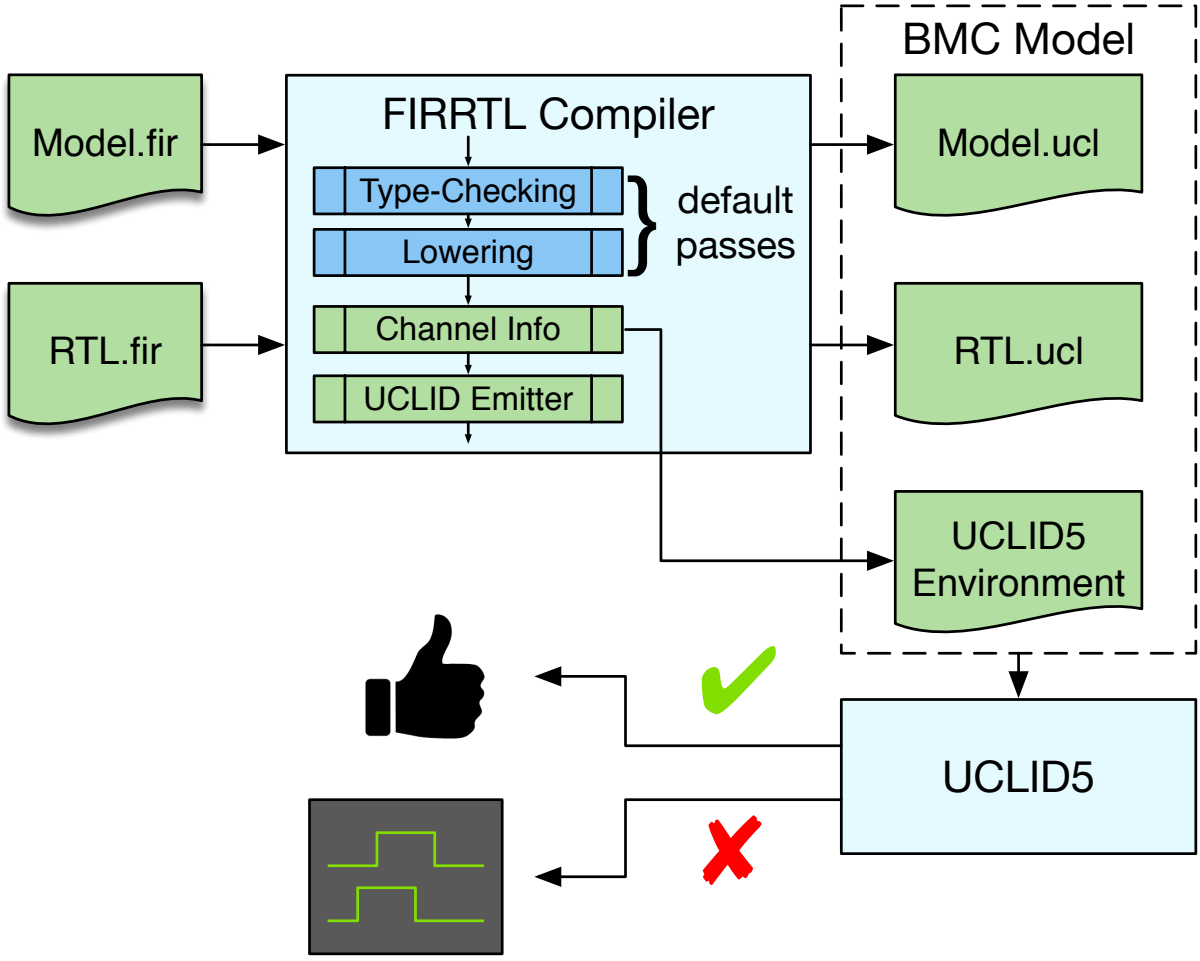

Figure 9.1: LIME Flow

9.2 Model Checking LI-BDNs

In all LIME property checking flows, the general structure of the model resembles the diagram shown in Figure 9.2. Here, the system is the LI-BDN that simulates a given reference RTL component, and the environment is the set of sources that generate input tokens for the LI-BDN and the set of sinks that consume output tokens. In this section, "simulation LI-BDN" is used in lieu of "simulation model," to avoid confusion with the "model" from model checking.

When checking PI, NED, and SC properties of simulation LI-BDNs, the environments always model the sources and sinks as abstract queues. Input queues nondeterministically present tokens to the simulation LI-BDN and track the number of consumed tokens (dequeue count). Output queues use *credits* rather than a finite capacity, with a credit nondetermin-

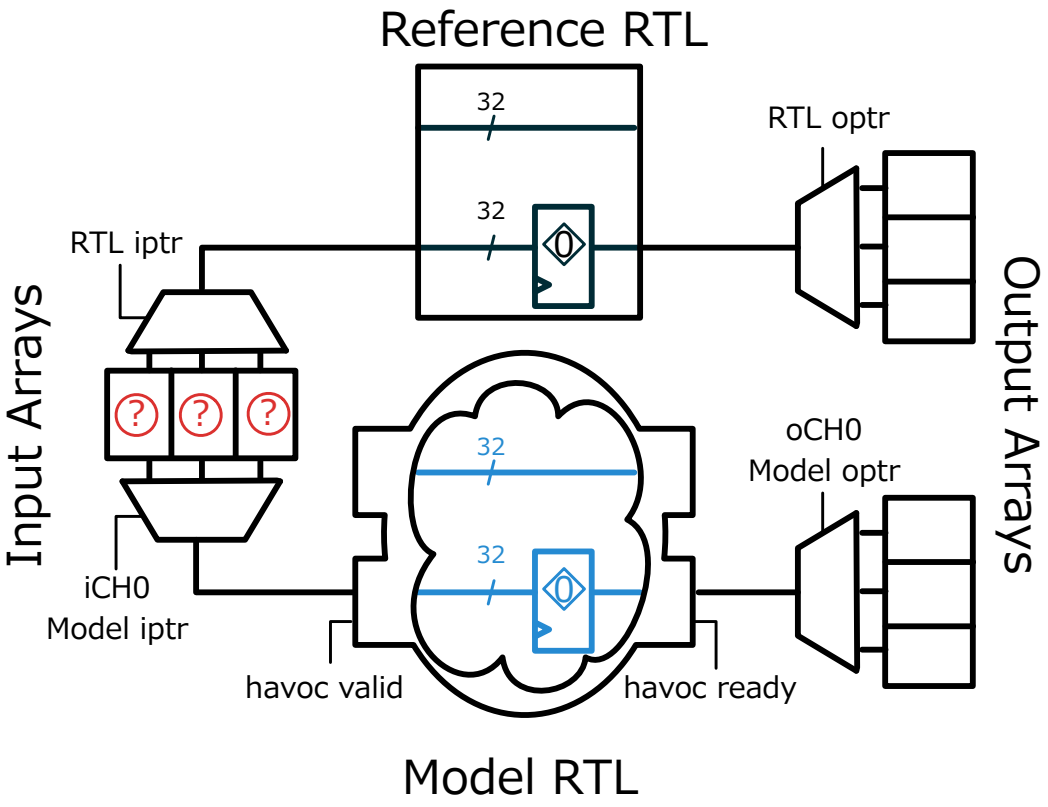

Figure 9.2: Partial Implementation Model

istically being added each step. The advantage of this is that it allows for arbitrary token arrivals and output back-pressure while offering guarantees that are not respected by randomizing inputs to the LI-BDN; specifically, validity of source data and readiness of sinks are *stable*, meaning that a source will not cease to have a valid token if it is not consumed and a sink will not cease to be ready if it is not provided with a token.

9.2.1 Partial Implementation

PI guarantees that the behavior of the simulation LI-BDN will be a *cycle-exact* representation of a particular Synchronous Sequential Machine (SSM), if its environment is itself a cycle-exact simulation of the inputs of the SSM. Formally, Vijayaraghavan et al. [89] define PI as:

A BDN R partially implements an SSM S iff

1. There is a bijective mapping between the inputs of S and [the input tokens of] R, and a bijective mapping between the outputs of S and [the output tokens of] R.

2. The output histories of S and R match whenever the input histories match, i.e.,

$$\begin{aligned}&\forall n > 0\\ &\quad I(k) \text{ for } S \text{ and } R \text{ matches } (1 \le k \le n)\\ &\Rightarrow O(j) \text{ for } S \text{ and } R \text{ matches } (1 \le j \le n)\end{aligned}$$

To provide matching input histories and compare output histories, the PI model composes the simulation LI-BDN with the reference SSM RTL. For each input channel, a nondeterministic sequence of input values is provided to the two implementations: as synchronous, cycle-by-cycle signal for the SSM, and as an abstract source FIFO model for the LI-BDN. On the output side, abstract sinks record the output token histories of the SSM, which are then compared with the cycle-by-cycle output histories of the SSM.

Using this construction, the environment forces the input histories of the LI-BDN and the SSM to match, while capturing their output histories. In this environment, we define PI as a conjunction of invariants, each ensuring for some output o_j that the output histories of the SSM and LI-BDN match according to correspondence operator $\widehat{=}$ based on the bijection between output tokens and outputs. Here, the property being verified must assert the conjunction of the PI invariant PI_j for every $o_j \in O$.

Invariant PI_j

$$\forall i \in [0, \text{cycles})\ i < \text{enq\_cnt}_j \Rightarrow \text{SSM\_hist}_j(i) \mathrel{\widehat{=}} \text{BDN\_hist}_j(i)$$

9.2.2 No Extraneous Dependencies

Vijayaraghavan et al. [89] formally define the No Extraneous Dependencies (NED) property:

> A primitive BDN has the NED property if all output FIFOs have been enqueued at least $n-1$ times, and for each output O_i, all the FIFOs for the inputs in `CombinationallyConnected`(O_i) are enqueued n times, and all other input FIFOs are enqueued at least $n-1$ times, then O_i FIFO must eventually be enqueued n times.

This definition relies on the definition of `CombinationallyConnected`(O_i), which is a set of input signals associate with each output O_i in the original SSM. This set is the transitive closure of combinational connections in a circuit, where a combinational connection is the use of a value of a signal within the same cycle, such as that of a wire or intermediate expression, but not that of a state element such as a register or memory. Throughout this section, we use the notation $\mathbf{CC}(O_i)$ to denote such a set associated with output O_i.

To express the NED property in a form amenable to automated checking, we represent it as the conjunction of multiple LTL properties, each of which enforces that a particular output o may have no extraneous dependencies. Here, $n-1$ from the above description

corresponds with the minimum number of tokens enqueued by any output channel of the LI-BDN; therefore, the property expresses an obligation for o to produce an output when at least n tokens have arrived at all the inputs to which o is combinationally connected, at least $n-1$ tokens have arrived at all other inputs, and no more than $n-1$ tokens have been produced by o. This constraint on o may then be expressed as an LTL property.

NED LTL property for output o

$$\begin{aligned}
\mathbf{CC}_j(i) &:= \text{ output } o_j \text{ depends combinationally on input } i \\
\text{obligated}_j &:= \min_{\{i \in I : \mathbf{CC}_j(i)\}} \text{enq_cnt}_i > \min_{\{o \in O\}} \text{enq_cnt}_o \\
&\quad \bigwedge \quad \min_{\{i \in I\}} \text{enq_cnt}_i \geq \min_{\{o \in O\}} \text{enq_cnt}_o \\
&\quad \bigwedge \quad \text{enq_cnt}_{o_j} = \min_{\{o \in O\}} \text{enq_cnt}_o \\
\text{NED}_j &:= \mathbf{G}\left(\text{obligated}_j \Rightarrow \mathbf{F}\left(\text{out_ready}_j \,\mathbf{R}\, \text{out_valid}_j\right)\right)
\end{aligned}$$

9.2.3 Self-Cleaning

Vijayaraghavan et al. [89] define SC:

> A primitive BDN has the SC property, if when all the outputs are enqueued n times, all the input FIFOs must [eventually][1] be dequeued n times, assuming an infinite source for each input.

As with PI and NED, the SC property can be expressed as a conjunction of LTL properties, each specifying when an input i is obligated to eventually dequeue a token. A common term in all of the properties is the minimum number of enqueued tokens by any output port; this value corresponds with n in the English-language description of the property. As part of the LTL property for input channel i, we add a signal obligated_i indicating that the simulation LI-BDN has dequeued fewer than n tokens from that channel.

SC LTL property for input i

$$\begin{aligned}
&\text{obligated}_i \Leftrightarrow \text{ input channel } i \text{ has a dequeue obligation} \\
&\text{obligated}_i := \text{deq_cnt}_i < \min_{\{o \in O\}} \text{enq_cnt}_o \\
&\text{SC}_i \ : \mathbf{G}\left(\text{obligated}_i \Rightarrow \mathbf{F}\left(\text{in_valid}_i \,\mathbf{R}\, \text{in_ready}_i\right)\right)
\end{aligned}$$

[1] Clarified in [88].

9.3 Verifying Multi-Ported Memory Models with LIME

The LIME flow can be applied to any GOLDEN GATE simulation model, but is especially useful for the widely applicable memory optimization models discussed in Chapter 5. While the optimized simulation model is the output of a generator that is parameterized in port count and memory size, checking a subspace of the optimized models using LIME provides a high degree of confidence in the correctness of the transformation. In multi-core SoC, this overhead is also be amortized across multiple identically parameterized memories.

From a usability perspective, LIME is an extremely convenient tool to find bugs in optimized simulation models of highly ported memories. For a decoupled simulation model, it is possible for implementation bugs to appear only in very specific corner cases, such as a certain interleaving of I/O token arrivals interacting pathologically with the write collision semantics of the memory. Instead of requiring the effort of a thorough, model-specific directed random test, LIME offers a push-button bounded model check that finds all bugs that can manifest within the time horizon of the bound. While a 20-cycle BMC bound has sufficient depth to cover the full space of I/O token arrival interleavings over several target cycles, Table 9.1 shows that this bound results in quick runtimes; longer BMC checks can be amortized over many uses of common configurations.

Register file	Parameters	PI BMC Runtime (s)
Rocket integer	31×64, 2R, 1W	445
Rocket FP	32×64, 3R, 2W	334
BOOM integer	100×64, 6R, 3W	637
BOOM FP	64×64, 3R, 2W	372

Table 9.1: Runtime for a bounded model check of 20 cycles for the Partial Implementation property of each optimized memory model.

Chapter 10

Conclusion

In this dissertation, we explore novel techniques to surpass traditional capacity limitations of FPGA-based simulators of digital systems. While existing techniques such as partitioning designs across multiple FPGAs provide a means to simulate large systems that cannot map to a single device, these techniques increase both effort and cost and significantly degrade simulation throughput. Instead, this work focuses on applying *compiler optimizations* to automatically translate designs to simulators with reduced resource utilization. Though these resource-utilization optimizations come at the cost of reducing the number of simulated cycles or time steps per second of real time elapsed on the host FPGA, they offer the potential to increase per-FPGA simulation capacity.

The optimizations presented in this work all derive from a simple tradeoff: rather than simulating a time step in a fully parallel manner, as with an FPGA prototype mapped directly from the RTL specification of the target design with an FPGA synthesis tool, a simulator may serialize aspects of this task over a—potentially variable—number of host FPGA cycles. This concept, known as *decoupling*, is borrowed from previous work on FPGA simulators such as HAsim and RAMP Gold. Decoupling opens the door to trading off space and time in the simulator implementation; in this respect, decoupled simulators are analogous to software RTL simulators and proprietary emulation systems based on specialized processors.

Unlike previous work in decoupled simulation, we aim to allow chip designers to generate optimized simulators directly from concrete RTL, without tying the implementation of the simulator to a particular target system. Therefore, we rely on an extensible *simulation compiler* as an implementation platform: the Golden Gate compiler, part of the FireSim open-source FPGA simulation framework. This compiler, collaboratively developed to support this dissertation and other research projects, allows a target design to be translated to a cycle-accurate simulator structured as a Latency-Insensitive Bounded Dataflow Network (LI-BDN). Such a network, consisting of *simulation models* communicating via asynchronous channels, provides a deadlock-free foundation for cycle-accurate simulation. Furthermore, Golden Gate provides interfaces to partition the target design to alter the topology of this network, allowing arbitrary components of the target design to be represented by dedicated, latency-insensitive models. In turn, this provides a high-level pattern for optimization:

individual components that represent fruitful targets for optimization due to their FPGA resource-utilization footprints are assigned to their own partition, and the resulting models are iteratively translated to slower but smaller implementations.

With these foundations in place, we present two optimizations, each offering the potential to significantly reduce the resource footprint of an FPGA simulator:

1. The multi-ported memory optimization targets memories that have complex port configurations, such as processor core register files. By simulating these memories with dedicated, multi-cycle models that can iteratively compute the behavior of even a combinational-read memory, it avoids the extreme resource cost of directly synthesizing FPGA implementations.

2. The instance multi-threading optimization targets repeated instances of a given module that collectively account for a large fraction of the simulator footprint. To reduce this cost, we model each instance via a thread of execution on a model that contains only a single copy of the module's combinational logic. This technique, inspired by previous threaded simulators such as ProtoFlex, RAMP Gold, and DIABLO, is realized for the first time as a transformation that can automatically thread simulators of arbitrary target RTL.

The two optimizations are evaluated by generating simulators of various configurations of the Rocket Chip SoC generator. In particular, we examine the ability of each optimization to extend the capacity of a Xilinx VU9P FPGA (hosted in the Amazon Web Services public cloud) to simulate a system with a larger number of cores. For this experiment, the systems instantiate a relatively large 5-issue configuration of the BOOM out-of-order core generator.

Each optimization effectively doubles the capacity of the VU9P as a host platform for simulating BOOM cores: while the baseline system can simulate only a two-core system, applying either optimization will allow a four-core configuration to fit. While this capacity increase comes at a performance cost, with throughput in cycles per second being reduced to 37% and 23% of the baseline two-core simulator, respectively, both still offer a previously impossible simulation capability at speeds competitive with partitioning.

While each optimization may be applied in isolation, they offer complementary benefits when applied to SoCs containing both repeated instances and large memories. Furthermore, the interleaved scheduling of the threaded simulation of the repeated instances offers the ability to hide much of the latency cost of the multi-ported memory optimization. By applying both optimizations, the capacity of the VU9P is extended to sixteen BOOM cores, a 700% improvement over the baseline. Though such a simulator achieves an effective target cycle throughput of only 1.56MHz, it represents an eight-fold increase in simulation "work" per cycle, and radically expands the capabilities of individual FPGAs.

Since the automated optimization passes in Golden Gate are generally applicable to any FIRRTL design with the appropriate architectural patterns, and since multiple instantiation of a large blocks is a common pattern in many systems, we further explore the use of instance threading to allow a large, multi-core accelerator to be simulated with a single

FPGA. In this system, each of a set of N accelerator cores pairs a small control processor with a systolic array co-processor aimed at linear algebra for machine learning. While only a single-core system will successfully map to a large Xilinx VU9P device with a standard compilation flow, the use of instance multi-threading on the accelerator cores extends the simulation capacity to accommodate an eight-core system. While this 700% increase in simulation capacity is paired with a corresponding slowdown in simulated cycles per second, the resulting simulator is able to model more "core-cycles" per second than the baseline. This result underscores the generality of Golden Gate optimizations and highlights their potential utility to a broad assortment of external users.

Finally, this dissertation focuses on exploring these optimizations as a research direction, significant engineering effort has been dedicated to producing a reliable tool that users may apply across many designs. To this end, we present two tools to detect bugs in complex Chisel- and FIRRTL-based projects. The first, an LTL property library for Chisel, allows designers to annotate their RTL with temporal property specifications, providing a low-overhead mechanism for verifying multi-cycle properties. The second tool, LIME, provides automated bounded model checking of correctness and deadlock-avoidance properties of LI-BDN simulators and their components. Both tools rely on a shared underlying infrastructure, including a FIRRTL backend that translates circuits to the language of the UCLID5 modeling system.

With these contributions, this dissertation provides a novel capability to automatically extend the simulation capacity of individual FPGAs. It provides a means to avoid the perilous "cliff" of exceeding the limits of available FPGA resources without undue effort by the end user. To this end, the optimizations and associated tools have been released as part of the open-source FireSim project. Ultimately, we hope that this work may ease the process of simulating and evaluating large systems and help productivity in digital hardware design.

10.1 Current Status and Future Work

As of this writing, both the multi-ported memory optimization and the instance multi-threading support have been incorporated into public FireSim releases. To better enable external users to apply these techniques to new target designs, examples demonstrating the frontend annotation API and the compiler settings have been added to the public web documentation. The LTL property toolkit and FIRRTL-to-UCLID5 backend discussed in Chapter 8 have also been released as part of the open-source ChiselUCL repository [21].

While the optimizations presented in this dissertation offer promising improvements to FPGA simulation capacity, there are many potential directions for future work. Most pressing is the need to apply these tools to a broader array of target designs; while this work focuses on BOOM-based Rocket Chip systems, the ability for FireSim users to enable the optimizations in their own designs will allow them to be applied to widely varying target designs. Further development will also likely include modifications to improve simulator performance. While the increase in FMR for the optimized simulators in 7 represents a

reasonable tradeoff when faced with the alternative of not fitting the design on the host FPGA at all, microarchitectural improvements could improve the relative performance of Golden Gate. In particular, the baseline primitive LI-BDN transformation limits the degree of decoupling allowed among the different channels of a model, which can degrade performance in a hybrid simulator. Future versions of Golden Gate could introduce more complex control logic to these models, allowing greater decoupling without prohibitive implementation cost.

While the Chisel LTL property library discussed in Chapter 8 has been released as an open-source tool, it is a relatively immature project. Given the high level of community demand for verification tools natively supporting Chisel-based designs, there is significant opportunity for continued development. Potential future work could include automatic runs of downstream tools like UCLID5, adding other property languages, or supporting different backends.

Bibliography

[1] Doug Amos, Austin Lesea, and Ren Richter. *FPGA-based Prototyping Methodology Manual: Best Practices in Design-for-Prototyping*. USA: Synopsys Press, 2011.

[2] Sameh Asaad, Ralph Bellofatto, Bernard Brezzo, Chuck Haymes, Mohit Kapur, Benjamin Parker, Thomas Roewer, Proshanta Saha, Todd Takken, and José Tierno. "A Cycle-Accurate, Cycle-Reproducible Multi-FPGA System for Accelerating Multi-Core Processor Simulation". In: *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2012, pp. 153–162.

[3] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

[4] Jonathan Babb, Russel Tessier, Matthew Dahl, Silvina Z. Hanono, David M. Hoki, and Anant Agarwal. "Logic Emulation with Virtual Wires". In: *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 16.6 (Nov. 2006), pp. 609–626. ISSN: 0278-0070. DOI: `10.1109/43.640619`. URL: `https://doi.org/10.1109/43.640619`.

[5] Jonathan Bachrach, Albert Magyar, Palmer Dabbelt, Patrick Li, Richard Lin, and Krste Asanović. "Cyclist: Accelerating Hardware Development". In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2017, pp. 1011–1018.

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. "Chisel: Constructing hardware in a Scala embedded language". In: *DAC Design Automation Conference 2012*. IEEE. 2012, pp. 1212–1221.

[7] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2017.

[8] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanović. "FASED: FPGA-Accelerated Simulation and Evaluation of DRAM". In: *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '19. Seaside, CA, USA: ACM, 2019.

[9] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. "Symbolic model checking without BDDs". In: *International conference on tools and algorithms for the construction and analysis of systems*. Springer. 1999, pp. 193–207.

[10] Randal E Bryant. "Simulation on a Distributed System". In: *Proc. of the 16th Design Automation Conference*. 1979, pp. 544–552.

[11] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. "Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions". In: *International Conference on Computer Aided Verification*. Ed. by E. Brinksma and K. G. Larsen. LNCS 2404. July 2002, pp. 78–92.

[12] *Cadence Palladium Z1 Enterprise Emulation Platform*. `https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/system-design-verification/palladium-z1-ds.pdf`. 2015.

[13] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. "Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection". In: *International Conference on Generative Programming and Component Engineering*. Springer. 2003, pp. 57–76.

[14] Luca Carloni, Kenneth McMillan, and Alberto Sangiovanni-Vincentelli. "Theory of Latency-insensitive Design". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20.9 (Nov. 2006), pp. 1059–1076. ISSN: 0278-0070. DOI: `10.1109/43.945302`. URL: `http://dx.doi.org/10.1109/43.945302`.

[15] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A Patterson, and Krste Asanović. "BOOM v2: An open-ource out-of-order RISC-V core". In: *First Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2017.

[16] Christopher Celio, David A. Patterson, and Krste Asanović. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*. Tech. rep. UCB/EECS-2015-167. EECS Department, University of California, Berkeley, June 2015.

[17] K. Mani Chandy and Jayadev Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations". In: *Communications of the ACM* 24.4 (1981), pp. 198–206.

[18] Leland Chang, Robert K Montoye, Yutaka Nakamura, Kevin A Batson, Richard J Eickemeyer, Robert H Dennard, Wilfried Haensch, and Damir Jamsek. "An 8T-SRAM for variability tolerance and low-voltage operation in high-performance caches". In: *IEEE Journal of Solid-State Circuits* 43.4 (2008), pp. 956–963.

[19] Kevin Cheang, Cameron Rasmussen, Dayeol Lee, David W Kohlbrenner, Krste Asanović, and Sanjit A Seshia. "Verifying RISC-V Physical Memory Protection". In: ().

[20] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. "Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE. 2007, pp. 249–261.

[21] *ChiselUCL: Utilities for generating UCLID5 models from Chisel and FIRRTL descriptions*. `https://github.com/uclid-org/chiselucl`. 2018.

[22] Eric S Chung, Eriko Nurvitadhi, James C Hoe, Babak Falsafi, and Ken Mai. "A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs". In: *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*. 2008, pp. 77–86.

[23] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. "Nusmv 2: An opensource tool for symbolic model checking". In: *International Conference on Computer Aided Verification*. Springer. 2002, pp. 359–364.

[24] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. "NuSMV: A new symbolic model verifier". In: *International conference on computer aided verification*. Springer. 1999, pp. 495–499.

[25] Katherine Compton and Scott Hauck. "Reconfigurable Computing: A Survey of Systems and Software". In: *ACM Comput. Surv.* 34.2 (June 2002), pp. 171–210. ISSN: 0360-0300. DOI: `10.1145/508352.508353`. URL: `http://doi.acm.org/10.1145/508352.508353`.

[26] Henry Cook, Wesley Terpstra, and Yunsup Lee. "Diplomatic design patterns: A TileLink case study". In: *1st Workshop on Computer Architecture Research with RISC-V*. 2017.

[27] Clifford E Cummings. ""full_case parallel_case", the Evil Twins of Verilog Synthesis". In: *Proc. SNUG Boston Meeting*. Citeseer. 1999.

[28] André DeHon. "Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100% LUT utilization)". In: *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field-Programmable Gate Arrays*. 1999, pp. 69–78.

[29] Shuwen Deng, Doguhan Gümüsoglu, Wenjie Xiong, Y. Serhan Gener, Onur Demir, and Jakub Szefer. "SecChisel: Language and Tool for Practical and Scalable Security Verification of Security-Aware Hardware Architectures". In: *IACR Cryptol. ePrint Arch.* 2017 (2017), p. 193. URL: `http://eprint.iacr.org/2017/193`.

[30] Brandon H Dwiel, Niket K Choudhary, and Eric Rotenberg. "FPGA modeling of diverse superscalar processors". In: *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE. 2012, pp. 188–199.

[31] Harry D Foster. "Trends in Functional Verification: A 2014 Industry Study". In: *Proceedings of the 52nd Annual Design Automation Conference*. 2015, pp. 1–6.

[32] *Gemmini, Berkeley's Systolic Array Generator*. `https://github.com/ucb-bar/gemmini`. 2019.

[33] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, Yakun Sophia Shao, Borivoje Nikolic, Ion Stoica, and Krste Asanović. "Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures". In: *arXiv preprint arXiv:1911.09925* (2019).

[34] Mike Gordon. "The Semantic Challenge of Verilog HDL". In: *Proceedings of tenth annual IEEE symposium on logic in computer science*. IEEE. 1995, pp. 136–145.

[35] David Grant, Chris Wang, and Guy GF Lemieux. "A CAD Framework for Malibu: An FPGA with Time-Multiplexed Coarse-Grained Elements". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 2011, pp. 123–132.

[36] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. "Understanding Sources of Inefficiency in General-purpose Chips". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10. Saint-Malo, France: ACM, 2010, pp. 37–47. ISBN: 978-1-4503-0053-7. DOI: `10.1145/1815961.1815968`. URL: `http://doi.acm.org/10.1145/1815961.1815968`.

[37] Scott Alan Hauck. "Multi-FPGA Systems". UMI Order No. GAX96-16615. PhD thesis. Seattle, WA, USA, 1995.

[38] William N.N. Hung and Richard Sun. "Challenges in Large FPGA-based Logic Emulation Systems". In: *Proceedings of the 2018 International Symposium on Physical Design*. ISPD '18. Monterey, California, USA: ACM, 2018, pp. 26–33. ISBN: 978-1-4503-5626-8. DOI: `10.1145/3177540.3177542`. URL: `http://doi.acm.org/10.1145/3177540.3177542`.

[39] "IEC 62531:2012(E) (IEEE Std 1850-2010): Standard for Property Specification Language (PSL)". In: *IEC 62531:2012(E) (IEEE Std 1850-2010)* (2012), pp. 1–184.

[40] "IEEE Draft Standard for VHDL Language Reference Manual". In: *IEEE P1076/D13, July 2019* (2019), pp. 1–796.

[41] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language". In: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), pp. 1–1315.

[42] "IEEE Standard for Verilog Hardware Description Language". In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), pp. 1–590.

[43] Accellera Systems Initiative. *Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual*. Jan. 2014.

[44] Intel. *Intel Stratix 10 Embedded Memory User Guide*. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-memory.pdf`. Aug. 2020.

[45] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Third Edition. Geneva, Switzerland: International Organization for Standardization, Sept. 2011.

[46] Adam Izraelevitz. "Unlocking Design Reuse with Hardware Compiler Frameworks". PhD thesis. EECS Department, University of California, Berkeley, Dec. 2019. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-168.html`.

[47] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations". In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2017, pp. 209–216.

[48] *JasperGold Property Synthesis Apps*. Aug. 2012.

[49] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 29–42.

[50] Richard E Kessler. "The Alpha 21264 Microprocessor". In: *IEEE Micro* 19.2 (1999), pp. 24–36.

[51] Mohammed A. S. Khalid and Jonathan Rose. "A Novel and Efficient Routing Architecture for multi-FPGA Systems". In: *IEEE Trans. Very Large Scale Integr. Syst.* 8.1 (Feb. 2000), pp. 30–39. ISSN: 1063-8210. DOI: `10.1109/92.820759`. URL: `http://dx.doi.org/10.1109/92.820759`.

[52] Donggyu Kim. "FPGA-Accelerated Evaluation and Verification of RTL Designs". PhD thesis. UC Berkeley, 2019.

[53] Donggyu Kim, Christopher Celio, Sagar Karandikar, David Biancolin, Jonathan Bachrach, and Krste Asanović. "DESSERT: Debugging RTL Effectively with State Snapshotting for Error Replays across Trillions of Cycles". In: *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*. 2018, pp. 76–80. DOI: `10.1109/FPL.2018.00021`. URL: `https://doi.org/10.1109/FPL.2018.00021`.

[54] Donggyu Kim, Adam Izraelevitz, Christopher Celio, Hokeun Kim, Brian Zimmer, Yunsup Lee, Jonathan Bachrach, and Krste Asanović. "Strober: fast and accurate sample-based energy simulation for arbitrary RTL". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2016, pp. 128–139.

[55] Helena Krupnova and Gabriele Saucier. "FPGA-Based Emulation: Industrial and Custom Prototyping Solutions". In: *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*. Ed. by Reiner W. Hartenstein and Herbert Grünbacher. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 68–77. ISBN: 978-3-540-44614-9.

[56] Murali Kudlugi, Soha Hassoun, Charles Selvidge, and Duaine Pryor. "A Transaction-Based Unified Simulation/Emulation Architecture for Functional Verification". In: *Proceedings of the 38th Annual Design Automation Conference*. 2001, pp. 623–628.

[57] Charles Eric Laforest, Ming G. Liu, Emma Rae Rapati, and J. Gregory Steffan. "Multi-ported Memories for FPGAs via XOR". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '12. Monterey, California, USA: ACM, 2012, pp. 209–218. ISBN: 978-1-4503-1155-7. DOI: 10.1145/2145694.2145730. URL: http://doi.acm.org/10.1145/2145694.2145730.

[58] Charles Eric LaForest and J. Gregory Steffan. "Efficient Multi-ported Memories for FPGAs". In: *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '10. Monterey, California, USA: ACM, 2010, pp. 41–50. ISBN: 978-1-60558-911-4. DOI: 10.1145/1723112.1723122. URL: http://doi.acm.org/10.1145/1723112.1723122.

[59] Chris Lattner and Vikram Adve. "Architecture for a next-generation gcc". In: *GCC Developers' Summit*. Citeseer. 2003.

[60] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[61] Kevin Laufer. *SMT Backend*. https://github.com/chipsalliance/firrtl/pull/1826. 2020.

[62] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. "Keystone: An open framework for architecting trusted execution environments". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

[63] Edward Lee and David Messerschmitt. "Synchronous data flow". In: *Proceedings of the IEEE* 75.9 (Sept. 1987), pp. 1235–1245. ISSN: 0018-9219. DOI: 10.1109/PROC.1987.13876.

[64] Charles E Leiserson, Flavio M Rose, and James B Saxe. "Optimizing synchronous circuitry by retiming (preliminary version)". In: *Third Caltech conference on very large scale integration*. Springer. 1983, pp. 87–116.

[65] Patrick Li, Adam Izraelevitz, and Jonathan Bachrach. *Specification for the FIRRTL Language*. `https://github.com/freechipsproject/firrtl/raw/master/spec/spec.pdf`. July 2020.

[66] Richard Lin, Bjoern Hartmann, and Elad Alon. "Improving Chisel Testing: Reusability Across Space and Time". In: *Chisel Community Conference*. 2018.

[67] Derek Lockhart, Stephen Twigg, Ravi Narayanaswami, Jeremy Coriell, Uday Dasari, Richard Ho, Doug Hogberg, George Huang, Anand Kane, Chintan Kaur, Tao Liu, Adriana Maggiore, Kevin Townsend, and Emre Tuncer. "Experiences Building Edge TPU with Chisel". In: *Chisel Community Conference*. 2018.

[68] Albert Magyar, David Biancolin, John Koenig, Sanjit Seshia, Jonathan Bachrach, and Krste Asanović. "Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes". In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2019, pp. 1–8.

[69] W. M. McKeeman. "Peephole Optimization". In: *Commun. ACM* 8.7 (July 1965), pp. 443–444. ISSN: 0001-0782. DOI: `10.1145/364995.365000`. URL: `https://doi.org/10.1145/364995.365000`.

[70] Susan Owicki and Leslie Lamport. "Proving liveness properties of concurrent programs". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 455–495.

[71] Michael Pellauer, Michael Adler, Michel Kinsy, Angshuman Parashar, and Joel Emer. "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing". In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE. 2011, pp. 406–417.

[72] Michael Pellauer, Muralidaran Vijayaraghavan, Michael Adler, and Joel Emer. "A-Port Networks: Preserving the Timed Behavior of Synchronous Systems for Modeling on FPGAs". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 2.3 (2009), pp. 1–26.

[73] Gregory F Pfister. "The Yorktown Simulation Engine: Introduction". In: *19th Design Automation Conference*. IEEE. 1982, pp. 51–54.

[74] Amir Pnueli. "The Temporal Logic of Programs". In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS '77. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57. DOI: `10.1109/SFCS.1977.32`. URL: `https://doi.org/10.1109/SFCS.1977.32`.

[75] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. "Global value numbers and redundant computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1988, pp. 12–27.

[76] Graham Schelle, Jamison Collins, Ethan Schuchman, Perrry Wang, Xiang Zou, Gautham Chinya, Ralf Plate, Thorsten Mattner, Franz Olbrich, Per Hammarlund, Ronak Singhal, and Jim Brayton. "Intel Nehalem Processor Core Made FPGA Synthesizable". In: *Proceedings of the 18th annual ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2010, pp. 3–12.

[77] Colin Schmidt, Alon Amid, John Wright, Ben Keller, Howard Mao, Keertana Settaluri, Jarno Salomaa, Jerry Zhao, Albert Ou, Krste Asanović, and Borivoje Nikolic. "Programmable Fine-Grained Power Management and System Analysis of RISC-V Vector Processors in 28-nm FD-SOI". In: *IEEE Solid-State Circuits Letters* 3 (2020), pp. 210–213.

[78] Sanjit Seshia and Pramod Subramanyan. "UCLID5: Integrating Modeling, Verification, Synthesis and Learning". In: *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. Oct. 2018, pp. 1–10. DOI: 10.1109/MEMCOD.2018.8556946.

[79] *Setting a New Lint Benchmark*. https://www.realintent.com/setting-new-lint-benchmark.

[80] Ofer Shacham, Omid Azizi, Megan Wachs, Wajahat Qadeer, Zain Asgar, Kyle Kelley, John P Stevenson, Stephen Richardson, Mark Horowitz, Benjamin Lee, Alex Solomatnikov, and Amin Firoozshahian. "Rethinking Digital Design: Why Design Must Change". In: *IEEE micro* 30.6 (2010), pp. 9–24.

[81] Tony Sloane. "Experiences with domain-specific language embedding in Scala". In: 2008.

[82] Stuart Sutherland. "I'm Still in Love with My X!" In: 2013.

[83] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. "RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors". In: *Proceedings of the 47th Design Automation Conference*. 2010, pp. 463–468.

[84] Zhangxi Tan, Andrew Waterman, Henry Cook, Sarah Bird, Krste Asanović, and David Patterson. "A case for FAME: FPGA architecture model execution". In: *Proceedings of the 37th annual international symposium on Computer architecture*. 2010, pp. 290–301.

[85] Russell Tessier and Heather Giza. "Balancing Logic Utilization and Area Efficiency in FPGAs". In: *International Workshop on Field Programmable Logic and Applications*. Springer. 2000, pp. 535–544.

[86] *The Veloce Strato Platform: Unique Core Components Create High-Value Advantages*. https://www.mentor.com/products/fv/request?selected=103372&null&fmpath=/products/fv/techpubs/requestpubs&id=103372. 2018.

[87] *UCLID5: formal modeling, verification, and synthesis of computational systems*. https://github.com/uclid-org/uclid. 2015.

[88] Muralidaran Vijayaraghavan. "Theory of composable latency-insensitive refinements". MA thesis. Massachusetts Institute of Technology, June 2009.

[89] Muralidaran Vijayaraghavan and Arvind. "Bounded Dataflow Networks and Latency-Insensitive Circuits". In: *2009 7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design*. IEEE. 2009, pp. 171–180.

[90] Angie Wang, Woorham Bae, Jaeduk Han, Stevo Bailey, Orhan Ocal, Paul Rigge, Zhongkai Wang, Kannan Ramchandran, Elad Alon, and Borivoje Nikolic. "A real-time, 1.89-GHz bandwidth, 175-kHz resolution sparse spectral analysis RISC-V SoC in 16-nm FinFET". In: *IEEE Journal of Solid-State Circuits* 54.7 (2019), pp. 1993–2008.

[91] John Wawrzynek, David Patterson, Mark Oskin, Shih-Lien Lu, Christoforos Kozyrakis, James C Hoe, Derek Chiou, and Krste Asanović. "RAMP: Research Accelerator for Multiple Processors". In: *IEEE Micro* 27.2 (2007), pp. 46–57.

[92] Michael Wehner, Leonid Oliker, and John Shalf. "Towards Ultra-High Resolution Models of Climate and Weather". In: *The International Journal of High Performance Computing Applications* 22.2 (2008), pp. 149–165.

[93] Henry Wong, Vaughn Betz, and Jonathan Rose. "Quantifying the Gap Between FPGA and Custom CMOS to Aid Microarchitectural Design". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.10 (Oct. 2014), pp. 2067–2080. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2013.2284281.

[94] Xilinx. *UG573: UltraScale Architecture Memory Resources User Guide*. https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf. Aug. 2020.

[95] Kenneth C Yeager. "The MIPS R10000 Superscalar Microprocessor". In: *IEEE Micro* 16.2 (1996), pp. 28–41.

[96] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanović. *SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine*. Tech. rep. Technical report, EECS Department, University of California, Berkeley, 2020.

[97] Victor Zyuban and Peter Kogge. "The energy complexity of register files". In: *Proceedings of the 1998 international symposium on Low power electronics and design*. 1998, pp. 305–310.